AD/A-000 085

FIVE LECTURES ON ARTIFICIAL INTELLIGENCE

Terry Winograd

Stanford University

Prepared for:

Industrial Science and Technology Agency
Advanced Research Projects Agency

September 1974

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

i

REPORT DOCUMENTATION PAGE		**READ INSTRUCTIONS BEFORE COMPLETING FORM**
1. REPORT NUMBER STAN-CS-74-459	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD/A-000 085
4. TITLE (and Subtitle) FIVE LECTURES ON ARTIFICIAL INTELLIGENCE		5. TYPE OF REPORT & PERIOD COVERED technical, Sept. 1974
		6. PERFORMING ORG. REPORT NUMBER STAN-CS-74-459 also AIM246
7. AUTHOR(s) Terry Winograd		8. CONTRACT OR GRANT NUMBER(s) DAHC 15-73-C-0435
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford University Computer Science Dept. Stanford, California 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS ARPA/IPT, Attn: S. D. Crocker 1400 Wilson Blvd., Arlington, Va. 22209		12. REPORT DATE Sept. 1974
		13. NUMBER OF PAGES 98
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) ONR Representative: Philip Surra Durand Aeronautcs Bldg., Rm. 165 Stanford University Stanford, California 94305		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Releasable without limitations on dissemination.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This publication is a slightly edited transcription of five lectures delivered at the Electrotechnical Laboratory in Tokyo, Japan from March 18 to March 23, 1974. They were intended as an introduction to current research problems in Artificial Intelligence, particularly in the area of natural language understanding. They are exploratory in nature, concentrating on open problems and directions for future work. The five lectures include: A survey of past work in natural language understanding; A description of the SHRDLU system; A comparison of representations used in A. I. programs; A rough sketch of some ideas for a new representation (cont'd)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) which combines features of the previous ones; A discussion of the applications of these ideas to programming systems.

ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM No. 246

SEPTEMBER 1974

COMPUTER SCIENCE DEPARTMENT REPORT
STAN CS 74-459

FIVE LECTURES on ARTIFICIAL INTELLIGENCE

Terry Winograd

Artificial Intelligence Project
Computer Science Department
Stanford University

September, 1974

ABSTRACT

This publication is a slightly edited transcription of five lectures delivered at the Electrotechnical Laboratory in Tokyo, Japan from March 18 to March 23, 1974. They were intended as an introduction to current research problems in Artificial Intelligence, particularly in the area of natural language understanding. They are exploratory in nature, concentrating on open problems and directions for future work. The five lectures include: A survey of past work in natural language understanding; A description of the SHRDLU system; A comparison of representations used in AI programs; A rough sketch of some ideas for a new representation which combines features of the previous ones; A discussion of the applications of these ideas to programming systems.

The research was supported in part by the Electrotechnical Laboratory, Pattern Information Processing System Project, and this document has been issued by them as PIPS-R-No.5. It was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract [DAHC15-73-C-0435]. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ARPA, or the governments of Japan and the United States.

Table of Contents

Lecture 1. 1
Computer systems for natural language

Lecture 2. 20
SHRDLU: A system for dialog

Lecture 3. 49
Representation: Formalisms for knowledge

Lecture 4. 64
Frames: Some ideas for a new formalism

Lecture 5. 83
Conceptual programming: Applying AI to program writing

Lecture 1

COMPUTER SYSTEMS FOR NATURAL LANGUAGE

In this first lecture, I will set some background for the rest of the course by describing a number of computer systems which have been built to work with natural language. These systems have been developed for English, but the ideas being developed are general and suited to any language. In fact, there are currently projects looking at a variety of languages to see the ways in which they are the same and the ways in which they present new problems. I hope you will keep this in mind throughout the lectures, although most of the examples I use will be from English.

Machine translation

I will begin by describing some of the history of natural language processing by computer, which goes back as far as the early 1950's. One of the first things people thought of when they built general purpose computers was the idea of using them to translate from one language to another. For the first fifteen years, translation was the focus of almost all computer natural language systems. Originally, people believed that translation involved two basic processes which might be called *dictionary* and *grammar*. A piece of text in one language could be translated by first looking each word up in the dictionary and finding the equivalent in the other language. The next step referred to the grammar -- the way the words were ordered, the endings and forms, etc. The ordering of the output would then be changed to correspond to the rules of the second language. The English sentence "I bought fish." can be analyzed as a subject followed by a verb followed by an object. In Japanese, you might say "sakana-o kaerimasita." in which the object comes first, and the subject does not appear explicitly unless a phrase is added like "watakusi wa", and the verb comes at the end. People believed that reasonably good translations could be done by treating these two operations of dictionary lookup and rearrangement separately.

It turned out that that approach did not work very well. When a person listens to language, he uses not just his knowledge of grammar, but also his knowledge of the world being discussed. The words and sentences have a meaning in the context of what the person is doing. Many sentences depend for their understanding on this use of knowledge. One simple example is the use of the English passive. If I have a sentence like: "The fish was bought by the cook." it translates into Japanese phrases containing "sakana-o" indicating that the fish is the object, and a phrase indicating that the cook was the actor -- the one who did the buying. If instead we have: "The fish was bought by the river." the structure is totally different. It was not the river who bought the fish, but it is a location and would be translated using a locational phrase with a Japanese word like "soba". Even though the two sentences look the same, we cannot decide on the translation by simply looking at the grammar. A person knows that a river cannot buy things -- it is not an appropriate action. In order to translate properly, the person knows facts about cooks and rivers, and fish, and what happens to them.

As a result of this problem it became obvious that a translation system could not succeed without trying to understand what it was translating. Only a small fraction of the sentences would be translated correctly. It isn't that the machine would fail to produce a sentence, but often the output in the new language would not convey the same meaning as the input in the original language.

In 1964, the National Academy of Sciences put out a report saying that although a tremendous amount of effort had gone into machine translation, it would not succeed using the techniques which had been developed. As a result, the attempt to build practical translation machines was delayed in hopes of coming first to a better understanding of the basic problems of language, grammar, and meaning. It is interesting to note that in the past few years people are again becoming interested in translation. They believe that some of the ideas which have been developed in the meantime may succeed in producing commercially useful translation systems. But it is still very much a hope -- it is not clear that we are yet at a point where a general translation device can be built.

Early AI systems

Following the end of the large translation projects, people turned instead to the problem of building computer systems which in some limited way tried to understand the language they dealt with. The most obvious way to judge understanding is to have a dialog. If you say something, and the person you say it to gives an appropriate response -- something which makes sense in the context of what you said -- then you believe he understood. If I ask you a question, and you give me an answer (even if you do not know the correct answer), I can usually tell whether you understood the question. If I request that you do something, and you do it or try to do it, then I can say you understood that request. If I give you a fact, and you later do something which depends on knowing it, I again have a test for your understanding.

STUDENT

People began to develop dialog systems involving an interchange rather than a body of text as dealt with in translation. One of the earliest systems which did question answering was a system named STUDENT, by Daniel Bobrow at MIT in 1964. It did simple word problems in algebra, from a high-school course. The input to the computer is a problem, like:

If the number of customers Tom gets is twice the square of twenty percent of the number of advertisements he runs, and the number of advertisements is 45, what is the number of customers Tom gets?

A human student would take this question, set up the equations, solve them, and give an answer like "162". The problem for the computer was to convert the language sentences into equations -- the test of understanding the sentences was producing the right equations to solve the problem.

This is very limited. If we use a phrase like "the number of customers Tom gets", then later ask "How many people buy things from Tom", the program has no way of connecting words like "customer" and "buy". The system used a simple kind of pattern match which looked for whole phrases corresponding to variables. For example it would set X = "the number of customers Tom gets" and could only match it to subphrases of that, like "the number of customers". Most of its knowledge was about the use of special mathematical words like "times", "of", "percent", etc. and it built the equations from patterns containing those. For a phrase like "20 percent of the number of customers" it set up an expression like ".2 x X", where X is the variable it assigned to "the number of customers". It recognized the use of the word "of" to represent this multiplication.

By doing this, the system performed fairly successfully on a set of simple problems. Its success depended on the fact that the problems did not involve interesting language deductions. However, it caused quite a bit of excitement, because it demonstrated that computers could solve problems involving language, even if in a small domain.

SIR

Another system done at about the same time was SIR, which stood for Semantic Information Retrieval, done by Bertram Raphael, also at MIT in Marvin Minsky's laboratory. This system answered simple questions about the relations between objects. You could type in sentences like:

A nose is part of a person.
A nostril is part of a nose.
A professor is a teacher.
A teacher is a person.

Then you could ask a question, like "Is a nostril part of a professor?" and it could perform simple logical operations involving transitivity and subset relations to answer "Yes." It handled only a limited number of possible relations, like "part of", "own", "has", and "is a kind of", and did not have a very general way of combining them. When more than one relation had to be combined to get an answer, special parts of the program had to take care of the interactions.

This system, like STUDENT, was written in LISP, and they were two of the first large LISP programs. The basic primitives of LISP are especially suited to the kind of symbolic non-numeric operations needed for these applications. For example, SIR used *property lists* to represent most of its knowledge. On the property list of the symbol PERSON, there would be a property named PARTS containing a list of parts like NOSE.

This program did not have a body of knowledge about the world built in. The only knowledge it had as part of its basic equipment involved these relationships like HAVE and PART. Specific facts about people and noses were typed in as English sentences like those above. It was extremely limited in the variety of sentences it could handle. Only a few simple patterns were recognized, like "A --- is a ---", or "A --- has a ---". It recognized 10 or 12 such patterns specifically, and any other sentences were not acceptable. The program was mostly concerned with the organization of the logical connections between the facts, rather than problems of grammar and language.

ELIZA

A very different kind of program done about the same time was ELIZA, by Joseph Weizenbaum. Weizenbaum asked the question "How could a system convince a person that it was carrying out intelligent dialog, without doing any more analysis of the language than absolutely necessary." He had the clever idea of considering the dialog between a psychiatrist and a patient. When you talk to a psychiatrist, he will often say things that don't seem directly related on the surface to what you said, but in trying to go below the surface he may have other connections in mind. This program is clever in that it does not have a deep analysis, but because you view it as a psychiatrist, you make the connections.

People can be quite convinced that this program is understanding a lot when in fact it has only a superficial recognition. A typical dialog might be: (the person's entries are in *italics*)

I am very lonely.
How long have you been very lonely?

This seems like an appropriate response. The system is looking for any pattern of the form "I am ---" and whenever it sees it, it types back "How long have you been ---", substituting the actual phrase. So if I type "I am therefore I think" it will say "How long have you been therefore you think". It does not try to make sense of the input -- only to find the pattern. Our dialog might continue:

Ever since my mother got disconnected.
Tell me more about your family.

Again this is triggered on a simple pattern. Whenever a phrase contains the word "mother", the program responds "Tell me more about your family." If you say "The prince's mother is the queen." it responds "Tell me more about your family." This is in fact very clever. A psychiatrist knows that a person who mentions mothers, whether his own or another, is very likely having subconscious thoughts about his own family. This is a deep kind of understanding which this program only mimics on the surface.

An important lesson to learn from this program is that we often read into a program's behavior our own ideas of what it understands. When it says something back, it may not have connected it in the way the speaker intended. This is a danger of judging any language program -- its success of understanding may not correspond directly to how good the response looks, since that response may have been generated by a special purpose trick.

Second Generation AI Systems

The three programs described above were the early group of language programs which convinced people that computer systems might treat language as something more than a string of symbols. They saw the meaning of sentences as something to be understood and manipulated. It was clear that they did not treat language in a sophisticated way. They all ignored the problems of grammar and complex structures which had been a main focus of the translation programs. They thought of meaning in a very limited way. SIR could only handle a small number of relations, STUDENT could only handle things whose meanings were equations, and ELIZA handled everything at the expense of not really trying to understand it.

Six or seven years later, a number of "second-generation" systems tackled a much larger range of the problems of language. They did this by restricting the things they talked about to a very narrow domain. Not narrow in the sense of being only algebra equations, but in the sense of choosing an area of the world involving a small number of objects and concepts.

LUNAR

One such system was Woods' LUNAR system which was developed over a period of time from 1967 to 1972. This system was designed to answer questions about the mineral samples brought from the moon by the astronauts. NASA has a large data base describing where each sample was found, what research has been done on it, what it is made of, and so forth. A geologist might sit down with a system like this and ask questions like "What is the average concentration of aluminum in high-alkali rocks?", and the system would answer back with a number like "8.45569 percent." In response to "How many breccias contain olivine?" it might respond "5", and in response to "What are they?" it would list the catalog numbers of those 5 samples.

Woods developed something called *transition net grammars* to describe for the computer the grammatical facts about English needed for interpreting complicated structures. The system uses the grammar to convert English sentences into requests in a special *query language* which is designed to interface with an information retrieval system he built for the large data base. We can think of his system as a translator from English into the query language (which is built in LISP). The result of typing a question is sending a request to this system, and getting back an answer. The range of what could be asked was fairly limited, as it knows only a small number of facts about each rock. There is a large number of samples, but only a few kinds of information. He also didn't worry about entering the data in English. All the information about the rocks is stored ahead of time in a special format, and the natural language capabilities are used strictly for understanding questions.

SHRDLU

At about the same time, I developed the SHRDLU program which converses with a person about manipulations it is performing in a simple world of toy blocks, like that used in the hand-eye robot projects at MIT and Stanford. There is a table with a set of simple objects like cubes and wedges, and an arm operates on those objects. It can pick them up and move them to build structures. The dialog is an interaction between the user and this program like the one to be shown in Lecture 2.

This system attempts to integrate all the aspects of language, in combining linguistic knowledge with a command of the world being discussed. The system answers questions, accepts commands, and takes in new facts as part of a dialog with a person, and there is a good deal of emphasis on the effects of context on understanding. It does not see each new sentence as an isolated problem, but keeps track of the actions, events, and sentences of the dialog to fit new facts into a meaningful context. Since this system will be described in detail in Lecture 2, I will defer discussion of it for later.

MARGIE

Another system developed in the early seventies is MARGIE developed by Schank's students (Goldman, Rieger, and Riesbeck) at Stanford. This system uses its knowledge of the structure of language to make paraphrases. In a natural language, the same thing can be said in many different ways. Different words and structures can convey the same meaning. MARGIE gives a series of different paraphrases of an input, demonstrating a type of understanding. In response to the input "John killed Mary by choking Mary", it produces a series of paraphrases like:

John strangled Mary.
Mary died because she was unable to inhale some air and she was unable to inhale some air because John grabbed her neck.

The program also draws simple inferences, so to an input like "John gave Mary some aspirin," it might respond:

John believes Mary wants an aspirin.
Mary is sick.

Rather than viewing this as a dialog system, we can better call it an *understander system*. Given a sentence input like the one to MARGIE there is no simple test of whether it understood. It is not like a translator where the output is either correct or wrong, or a question answerer where there is an appropriate response to a question or command. Instead, the result of an input sentence is a change to the program's internal knowledge structure. At some other point, you might test that knowledge structure by asking questions or presenting tasks, but the program is primarily concerned with the problem of taking in knowledge, rather than demonstrating it. The test then lies in the computer scientist looking at the structures it has built and seeing if they reflect a correct understanding. This can make it confusing to read about such recent programs, since there are no simple input-output criteria for understanding.

In the long run, such systems need to be able to make use of their information in interacting with the world. But in looking at current work on such systems we cannot be too rigid in our definition of "understanding". We must concentrate on the specific ways in which the system deals with knowledge.

Current Developments

The previous section describes most of the large well-developed systems which have been built. Most of the work going on now is much more fragmented. Rather than trying to put together a large system which carries out an entire dialog, people are concentrating on the components which must go into new systems. They represent sets of ideas for working with meanings, with the ultimate hope of combining these ideas in future large systems. The current projects on speech understanding (which I will describe later) are an exception to this, dealing explicitly with the problem of integrating the components of a total system.

The representation problem

The main problem people are facing might be called the *representation of knowledge*. Artificial intelligence in general might be characterized as the part of computer science concerned with this problem of representation. The early systems all had special forms of program and data to represent different sorts of meaning. STUDENT had a very simple idea of representation -- *meaning = equations*. That works only for things in a small mathematical domain. We cannot in general represent a fact as a simple equation. SIR had a simple representation using property lists, which could store other kinds of facts, but wasn't adequate for more complicated knowledge involving things like quantifiers. For these more complex interconnections between facts, we need a more general formalism. This is true both for understanding language and for other problem-solving and intelligent systems (like those for vision). People have developed a number of ideas for representations of knowledge in computer programs, which will be discussed in detail in Lecture 3. They form a basis for much of the work that is being done. In the rest of this lecture I will talk more specifically about those issues related to representing the information needed for natural language understanding and reasoning.

Case structures

One important idea which is being developed is that of *case structures*. The idea of cases in linguistics has been around for a long time. In learning a language like Latin, you learn the declension of nouns -- the set of cases which the word can take on. Fillmore (1968) developed this idea as a way of talking about grammars which do not have explicit marked cases like Latin. In an English sentence like "I bought the fish.", we can say that there is a case named AGENT for the object causing the action to happen, and another role PATIENT for the thing undergoing the action. There are other cases, as in the sentence "John gave Mary a bottle." John is the agent -- he did the act. The bottle is the patient, or thing acted upon, and Mary is the DATIVE or beneficiary -- the action was for her benefit. Fillmore's point is that a small number of such cases is sufficient for describing much of language structure. Different people have developed a variety of systems with different cases, but the important thing is that there are a small number of ways in which an object can be related to an action. Once we have named the 6 or 8 or 10 possibilities, this can be used to describe an action in computer memory. We specify the type of act, and the set of objects involved with the case they fill in. Simmons at the University of Texas has developed these ideas in a computer implementation. His program takes simple sentences and analyzes them into these deep case relations.

It is important to remember that this is a semantic, not a surface syntactic representation. In the sentence "I broke the dish.", I am the agent. In "The dish broke.", the dish is now the syntactic subject of the sentence, and on the surface level is filling the same place that "I" filled in "I broke the dish." But at the case structure level, the dish is the patient in both cases -- the thing acted upon - even though on the surface it is represented in different ways. It is interesting to compare different languages. In Japanese, I believe, the surface form is closer to these case structures than in English.

Larger scale structures

In going from translation systems to dialog systems, people expanded the set of issues they were looking at. In translation, they were concentrating on the dictionary and grammar. In dialog systems, there was a new emphasis on meaning, and the way knowledge is connected. But these systems still took a limited view of what was needed -- they dealt with language sentence by sentence, fact by fact. The concern was to take a particular sentence and relate it to a global data base or task. The problems were those of representing very specific facts, like "A nose is part of a person" or "I want you to pick up that block." Today, people are becoming aware of the ways in which this is too much of a limitation as well. We need to understand how the process of understanding can relate to larger structures of knowledge. In understanding a sentence, we are not simply taking in a new piece of cognitive structure, but making changes to larger structures which already exist. As part of your knowledge, you know a great deal about the kinds of connections which are possible between things. Programs must deal with sentences by fitting them into such a structure. We cannot view each new sentence as a "piece" which will eventually be combined with others to take an action or produce an answer. Instead when a new piece comes in, it is actively fit into existing structures, and a great deal of organization goes on long before any use is made of the information in answering a question or carrying out a command. There is an active process seeking to fit things together.

Conceptualizations

One example of larger conceptual structures is that of simple stories, like those studied by Chafe (1972). He talks about *conceptualizations*. A person who sets out to tell a story begins with a concept of the story as a whole. The problem is to convert that into a sequence of words, and the problem for the hearer is to deduce from those words what the structure is. So in looking at the process of story understanding, we need to analyze the structures. Figure 1.1 shows some of the structures used by Chafe. The first says that a fable is a story which illustrates some moral. Once we decide that we are hearing a fable, we will interpret what we hear as leading to some sort of moral, and the active search for that moral will affect the way we interpret the details of the story. A story in turn is made up of a world which serves as background for a plot. The plot involves a motivation which yields a sequence of events (the development) and the outcome of that is some sort of resolution.

This is an abstract description of the story. It is language, but not at the level of grammar. It deals instead with how we organize our thoughts. Because the understander has a knowledge of these structures ahead of time, he can fill in a number of extra connections between the pieces.

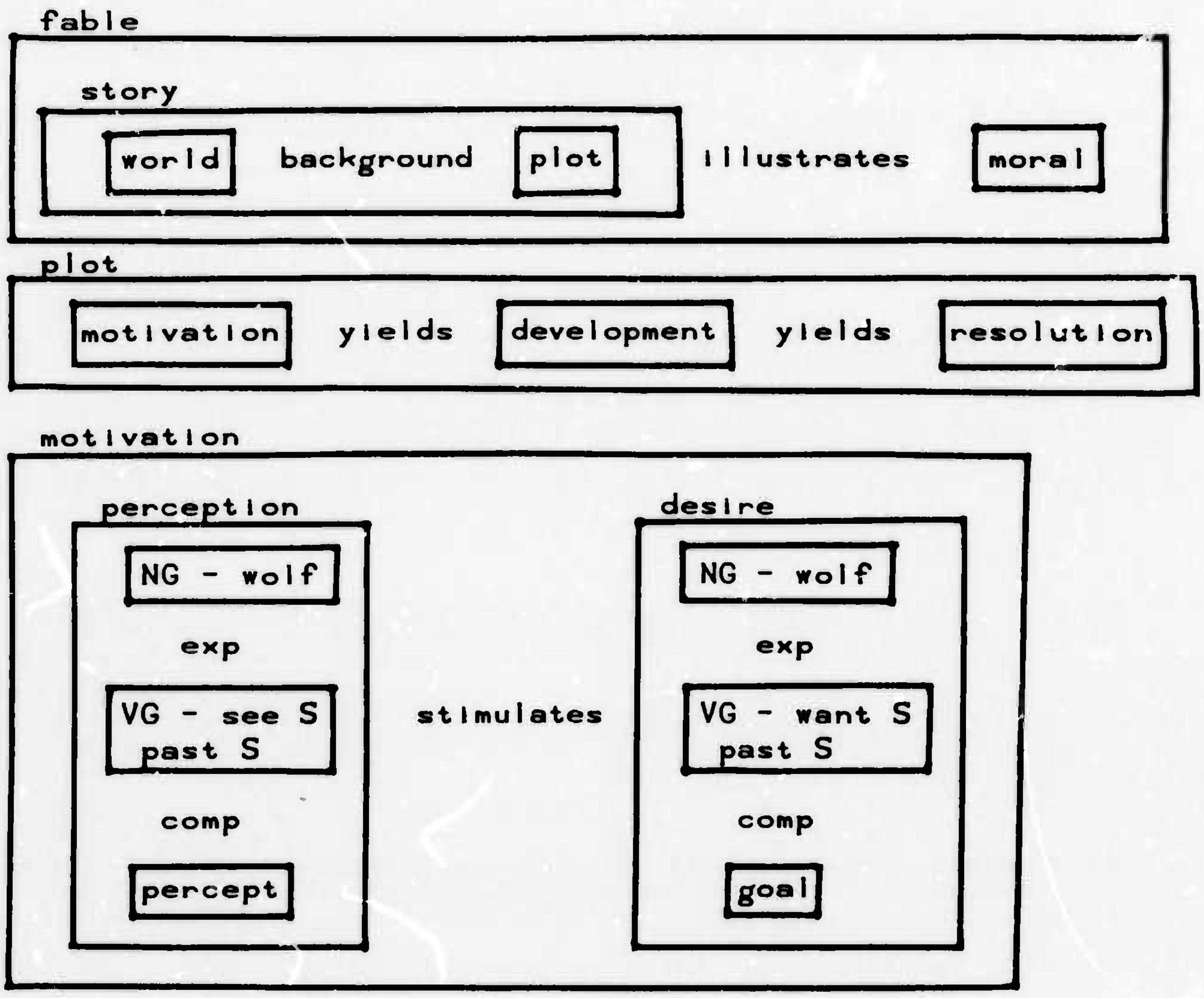

Figure 1.1. Conceptualizations for a simple fable (Chafe).

This particular analysis has not been explored very far in actual computer programs, but is an idea for some formalisms for these larger structures.

Scripts

Another formalism for story structures is Abelson's (1973) *scripts*. These also have not really been used in developed programs, but represent a very similar approach to looking for the larger cognitive structures for events. Abelson deals with those things which might be called "plot" in Chafe's work. He looks at the possible connections between events in a story, for example that a particular act may have been done to prevent some other event from happening. Others might include an act being done to make an event possible, one event causing another, etc. In his paper he presents a schema for a particular political ideology, which he calls a *Master script*. In this script (see Figure 1.2) there is a set of events and actions with relations to each other. This script is applied to a particular set of events to interpret the connections between them. When the person with this ideology reads about an event in a newspaper, he does not simply store away a fact like "There is a strike in Indonesia." He tries to fit it into his script by seeing it as a communist plot, a response to such a plot, and so on. In political analysis it is clear that different people can have very different schemata for viewing the same situations. By the time a fact gets stored in the knowledge structure, it has been changed by the way it fits into the structure. It is no longer "there was a strike", but "the communists are causing trouble" or "the people are responding to the oppression of the bosses." In fact, a person may later remember these without remembering any of the details of the specific act which took place.

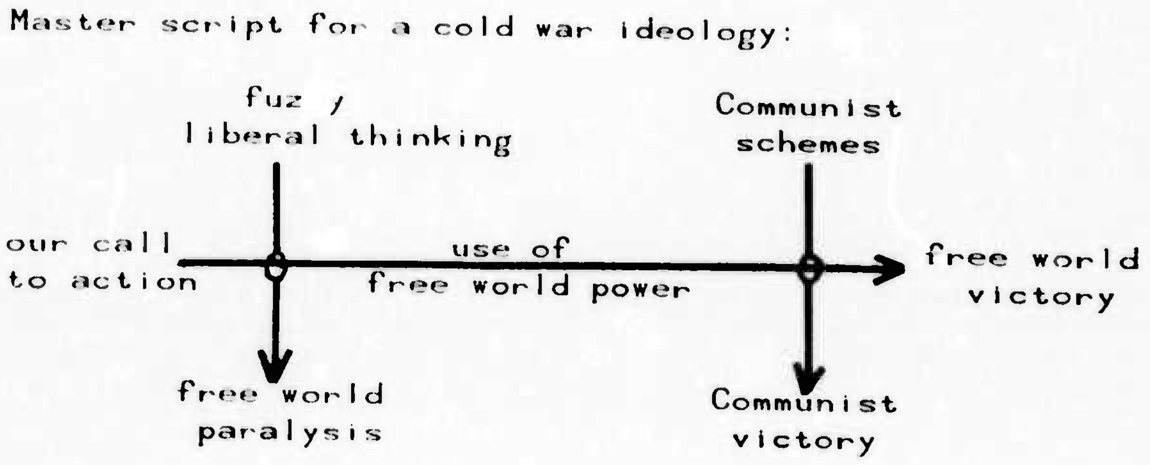

Figure 1.2. An ideological script (Abelson).

It isn't immediately obvious that this applies in areas less subjective than politics, but one important trend in natural language is the attempt to find this kind of "assimilation" happening in all sorts of understanding processes. When I make a statement of any sort, it isn't just stored, but you try to interpret it as fitting in to a larger "script". Programs are just now being built to try and do these things at many levels.

Conceptual dependency

Another system is Schank's *conceptual dependency* which he has been developing for a number of years. His point is that a single word in English is often a pointer to a complex conceptual structure. When you hear a word, you call forth a pattern for a structure and fill it out. Figure 1.3 shows a structure for "give".

John gave Mary a book.

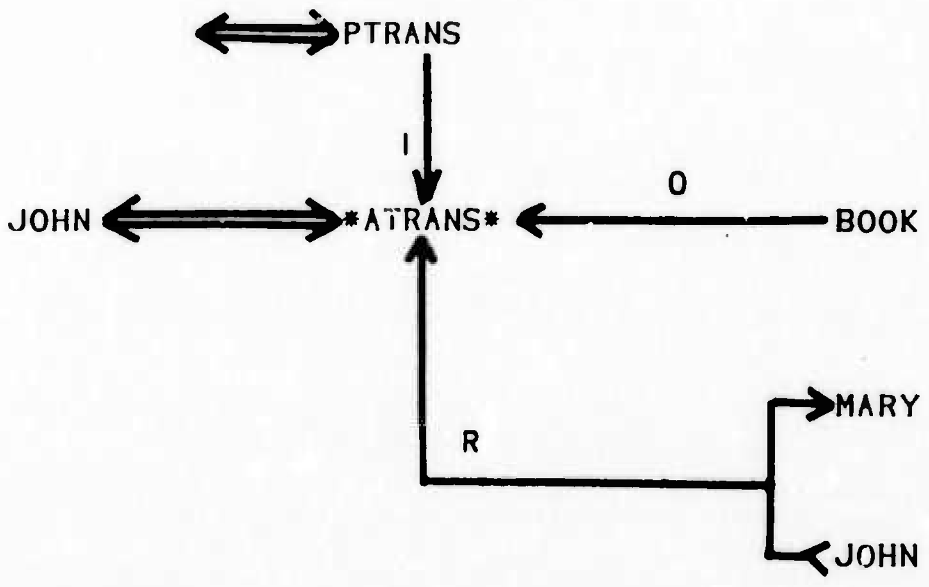

Figure 1.3. Conceptual dependency network for "give" (Schank).

If you hear that "John gave Mary a book." you will try to fill it out, including looking for facts about why and how the act was done. If we say "Sam grew corn" this is not the same as "Sam grew." It was the corn which did the growing, and Sam did some other act or acts to help it. Conceptual dependency makes this explicit in the structure, as illustrated in Figure 1.4 .

Sam grew corn.

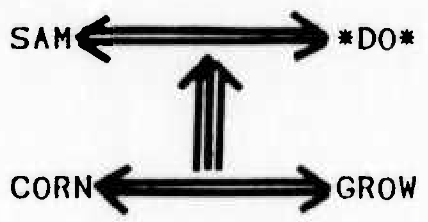

Figure 1.4. Conceptual structure for "Sam grew corn."

The multiple arrow indicates a cause -- Sam's doing something caused the corn to grow. In the English surface structure, "growing" seems a simple act, like "Sam hit the ball." But in the conceptual structure there is an element missing -- an unspecified act. Part of the design of a system must involve an awareness that things like this need to be filled in. We can look explicitly for the things Sam must have done.

Demons

Another mechanism for drawing connections between pieces of knowledge is Charniak's (1973) *demons*. He deals with the problem of recognizing that two facts should be connected. Once it is recognized that two facts are inter-related, they can often suggest a larger structure. His work deals with simple children's stories. We might have a sequence of sentences like:

Janie went to her piggy bank. She shook it. She took the nickel and went to the store.

A person hearing this sequence knows what happened. If we give it to a computer, then ask "Where did she get the nickel" there is no explicit answer available. There is no sentence saying that a nickel came out of the piggy bank -- only one saying she shook it, and another saying she took a nickel to the store. A person draws the obvious connection that the money must have come out of the piggy bank. This is a critical element of natural language. A person almost never provides all the detailed steps that connect the things he is saying. The listener knows enough to fill in a great deal of implicit information. If you ask me "What time is it?" and I respond "My watch is broken", that is fine -- I don't have to say "I cannot tell you because I do not know what time it is, because my watch is broken, and when a watch is broken it does not tell the correct time, etc." All that is obvious.

Charniak has taken some limited cases and used demons to make those connections. When his system notices that Janie is shaking a piggy bank, it sets up a demon which states "If you see any mention of money in the next few sentences, it probably came from the piggy bank." We can think of this demon as sitting in the program looking for mentions of money, waiting to jump out and say "Aha! I know where that came from." There are lots of complicated problems in making use of demons, since there will often be a number of them, looking for different things, and their interactions lead to many difficulties. But there is an important idea in the explicit setting up of expectations which may be met by further bits of the input.

Rings

Another paper by McDermott (1973) proposes *ring structures* for connecting sequences of hypotheses and events. As facts are described to the system, it looks for connections between them. If it is told that "Fred is a bird." and "Fred can't fly." it will not simply store away those facts, but look for a third fact which makes them plausible. The two original facts set up a *tension* since they normally don't go together, and the tension is resolved by constructing a ring with another fact which the system hypothesizes. One solution is to infer that a flightless bird must be a penguin. A natural inference from being a penguin would be that he lives in Antarctica, so if the system now learns that "Fred lives in Sydney", we must find some new fact to resolve the tension --

perhaps he lives in a zoo. It is not possible to be sure what the right thing is to fill in the ring -- if instead of hypothesizing that Fred is a penguin we had hypothesized that he is a dodo bird, that would also resolve the tension of a flightless bird, but it would then not take an extra ring to fit together with living in Australia. The program is faced with having to hypothesize plausible facts to make things fit together, but has to be willing to go back and change its mind -- to dissolve the ring and reconnect it another way. Of course before it discovers the problem, it may have used its hypothesized facts to build still other rings.

McDermott's system accepted a simple sequence of events concerning a monkey in a room with some simple objects. His system was not in the traditional AI position of simulating the monkey's problem solving, but in the position of an observer, trying to understand its actions. If the monkey picks up a banana, the program needs to explain why it would do such an act. It makes the obvious hypothesis that the monkey is hungry. If the monkey then sets the banana down, this creates a tension with the fact that it wants to eat it. Either a new ring must be constructed with a further fact (maybe there was a sudden noise), or the old one must be dissolved, and the system must find another reason for picking up the banana -- perhaps the monkey was just curious. McDermott was concerned primarily with structuring sets of facts and contexts to make this kind of process possible.

Speech Understanding Systems

Most of the problems described above might be characterized as relating to the *chunking* of knowledge. Such ideas are in a beginning stage, and are not yet integrated into systems. Another aspect of current work involves the organization of integrated natural language systems. It is particularly active in the context of a number of speech understanding systems. A concerted research effort was begun 2 years ago by the Advanced Research Projects Agency (Newell et.al. 1973) to produce in a 5 year period a working speech system. A person could converse with this system in normal connected English. This is quite different from the existing programs to represent isolated spoken words, as it must combine all different aspects of language. There is a great deal of emphasis on avoiding the "brittleness" of current systems. This is the quality, very common in computer systems, that they can't bend at all without breaking. If there is an input which is almost what the system would expect, but not quite, the system does not just find the input more difficult to handle, but fails completely. If the user types in one word or punctuation mark wrong, or the system is missing one fact, it is not flexible enough to say "Oh, that really must have been..." Much of the emphasis on larger scale organization is to get this kind of flexibility. Since a human understander is interpreting an input in the context of larger structures, he may be willing to interpret it in a way quite different from what it looks like if that is what is needed to make it comprehensible -- he is willing to bend it quite a way to make it fit. In speech, this is an immediately necessary feature, since actual acoustic signals cannot be processed to provide enough information to provide the kind of certainty a brittle system would need.

If the context provides only a few choices of what a word must have been, it is often possible to decide which fits the acoustic signal best. But faced with the task of scanning an utterance for words, there is no way to do it accurately and completely. So the system has to be organized to be very flexible and forgiving about what it finds. It has to know what it is looking for on the basis of larger structures and use the input as a test. There are a number of different organizational schemes being developed. It is important to note that these organizational ideas are not needed only for speech systems -- as language programs of any sort try to handle less constrained inputs, they must share many of the same features.

Linear organization

The simplest form of organization is that shown in Figure 1.5 . To some large extent, the different facets of our language knowledge can be separated into components, and we can organize a system by building them separately and letting the data flow from one to the other.

$$\begin{tikzcd}
\text{WORD EXTRACTION} \arrow[d] \\
\text{SYNTAX} \arrow[d] \\
\text{SEMANTICS} \arrow[d] \\
\text{REASONING}
\end{tikzcd}$$

Figure 1.5. Linear organization for a speech understanding system.

The first component needs to determine the words in the input. In the case of written language this involves only a dictionary lookup to see what character strings are words. For speech, this stage is much harder, involving a good deal of acoustic and phonetic processing.

The next stage is *syntax*. It handles the facts about the arrangements of words which are usually included in a grammar. If it sees the two sentences "The dog bit the man." and "The man bit the dog." the words are all the same, but the ordering gives more information. The syntactic component involves an operation of *parsing* which takes the word sequence and builds grammatical structures.

The next stage is often called *semantics* and describes the connections between these syntactic structures and meaning. If we have a structure like "I ran", I am the actor -- I did the running. If we have "The dish broke", even though the dish is the subject it is not the actor, but rather, something happened to it. Things which have the same syntactic structures on the surface can convey different meaning. The semantics component then needs to convert the linguistic structures into some sort of internal representation that can be used for reasoning. The final stage is a reasoning component

Most of the complete systems which have been built operate in a simple way. They assume that you can first find the words in a dictionary, then call a parser which analyzes the syntactic structures, then call a semantic analyzer which converts it into the meaning representation, and finally call on the set of reasoning programs which use it.

One system (developed at the Stanford Research Institute) tries to reverse this directionality. In speech, it isn't easy to say what word is appearing in the wave form, so they try to predict by coming from the other end of the chain. They ask "What do I think is being said (based on reasoning)?" followed by "How would that be phrased (syntax)?" and finally "Therefore what words do I expect to see?" In some cases this approach works well, but it depends on there being a very limited set of possible messages. If I ask a question which is to be answered "yes" or "no," then it is easy to recognize the answer. On the other hand, If you walk in and begin a lecture, it is not the case that I have a clear expectation for the words you will begin with. Often we don't have anywhere near enough information to go look for a particular word which might have been said.

Heterarchical organization

Another style of organization is the *heterarchical* system. In this kind of system, there are a number of sub-components working together, and any one of them can pass information to any other one directly. There is no strict chain of command, progressing from one stage to the next. Any component can be called at any time if its knowledge seems likely to be useful. In the system at Bolt, Beranek, and Newman (Woods, 1972) the components are as shown in Figure 1.6 .

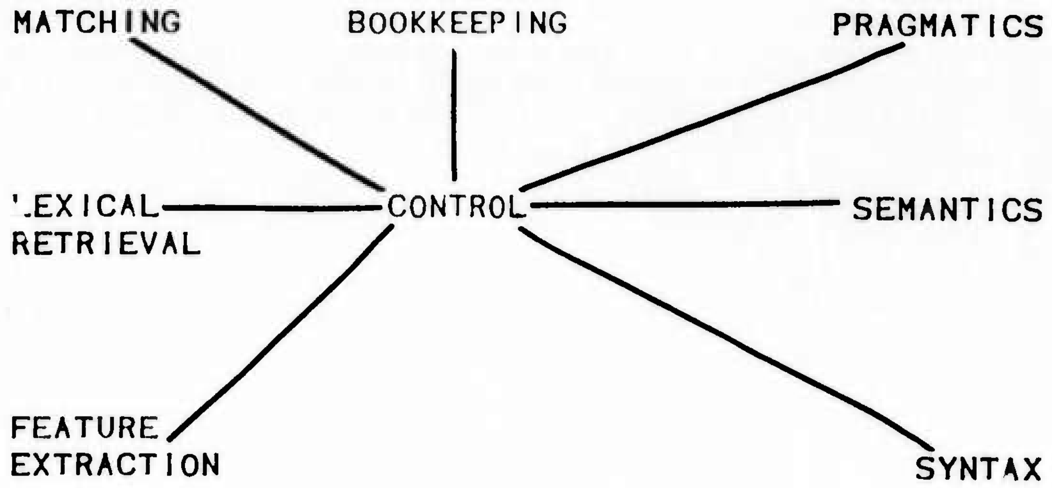

Figure 1.6. Heterarchical system organization.

The feature extractor looks at the incoming wave forms, and suggests possible phonetic features. These then can be used by the lexical retrieval component to see what words are possibly there. The matching component takes a possible word and checks it against a piece of the wave form to check the likelihood that it is actually there. The syntax component knows the syntactic structure of language and uses this to evaluate possible sequences of words. The semantic component knows about simple semantic facts (like our early example that "rivers can't buy things"), and finally a pragmatic component does the reasoning about the specific topic being discussed. There is also a bookkeeper to keep track of the possible analyses being considered, and a control box to decide which component will do what when. Of course the interesting structure from an organizational viewpoint lies in this control box.

The particular method used to understand the complexities of control is called *incremental simulation*. Given a number of different types of knowledge, the problem is to understand when to use each one, and what sort of information needs to be passed from one to the other. At one time, the syntax may be used to suggest what words are present, and the matcher would check to see how plausible they are. In another case the feature extractor might find a set of words with a great deal of certainty, and the syntax would have to find a way to analyze their structure. Woods' group simulated an understanding system by having people doing each of the component tasks, and using the computer to interface between them. This allowed the entire system to be working even though the details of each component were not worked out. By studying the kinds of interactions which were necessary to put together the knowledge of the 6 individuals simulating the components, they are figuring out how to organize the control component. Gradually programs will replace the person doing each component task, within the context of a clearer idea of how it should be fit in -- what sort of information it can expect to get, what it can be expected to produce, and when it should be used.

Pandemonium

The heterarchical system assumes that the builders of a system must develop a complex control structure suited to the task. The result of this research is expected to be a detailed understanding of just how the tasks interact and what should be done when. A very different model (which I am calling *pandemonium*) is being tried by the group at Carnegie-Mellon University (Reddy 1973). The idea is to avoid worrying about complicated control structure. Rather than having to anticipate which sort of information should be used when, the subsystems are all allowed to work independently and share information through a *hypothesis scratchpad*, as shown in Figure 1.7.

It is as if we had a group of experts working on a common task but no one of them knew anything about the others. Each expert might not even know how many others there were, or what kind of things they dealt with. Communication is managed by having each expert know how to propose sequences of words, and assign a degree of confidence to them. Any individual component can look at the current set of hypotheses, and either add new ones to it, or change the level of confidence in one of the old ones, depending on whether it makes sense with respect to his particular knowledge. Thus the *overlord* has no authority to decide what is to be done, or what knowledge is to be passed, but is just a bookkeeper. This sort of system has the virtue that it is easy to put in new components, and make major changes to their functions, without disturbing the rest of the system. It is highly modular. The interesting question is whether such a system can be successful without the more tailored interactions of a heterarchical system.

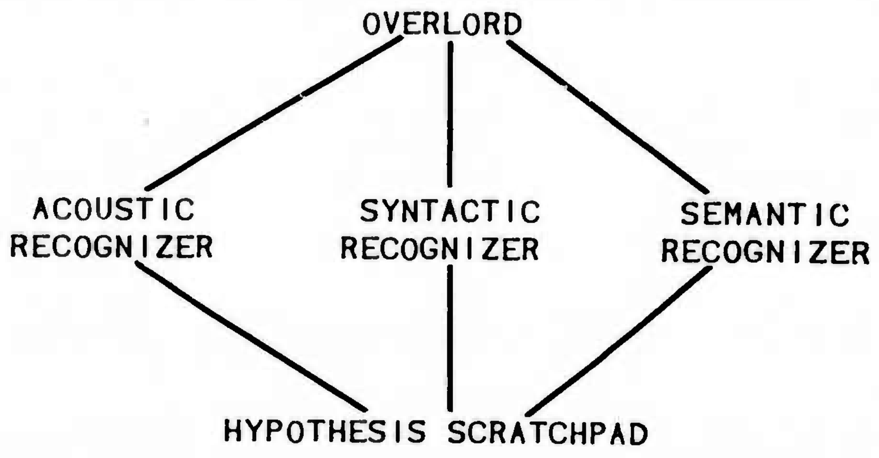

Figure 1.7. The pandemonium organizational model.

Summary

In summary, the main problems being faced in language systems today are "How can we make use of larger structures of knowledge to help in understanding?" and "How can a system be organized to make flexible use of a variety of different sorts of knowledge?" Within the next few years we will see large systems incorporating many of the ideas now being developed, and we can expect them to perform significantly better than the ones which are now available.

References for lecture 1

Machine Translation

Yorick Wilks, An artificial intelligence approach to machine translation, in Schank and Colby (eds.), *Computer models of thought and language*, pp. 114-151.

Early AI systems

Daniel Bobrow, Natural language input for a computer problem solving system, in Minsky (ed.), *Semantic information processing*, pp. 134-215.

Marvin Minsky (ed.), *Semantic information processing*, MIT Press, 1968.

Bertram Raphael, SIR: A computer program for semantic information retrieval, in Minsky (ed.) *Semantic information processing*, pp. 33-133

Joseph Weizenbaum, ELIZA, *Communications of the ACM* 1966, 9, pp. 36-45.

Second-generation systems

MARGIE: Memory, Analysis, Response Generation, and Inference on English, Advance papers, *Third International Joint Conference on Artificial Intelligence*, pp. 255-261.

Terry Winograd, *Understanding natural language*, Academic Press, 1972.

William Woods, R.M. Kaplan and B. Nash-Webber, The lunar sciences natural language information system: final report, BBN Report 2378, Bolt Beranek and Newman, Cambridge, Mass., 1972.

Case structures

Charles Fillmore, The case for case, in E. Bach and R. T. Harms, (eds.), *Universals in linguistic theory*, New York: Holt, Rinehart and Winston, 1968, pp. 1-68.

Robert Simmons, Semantic networks: their computation and use for understanding English sentences, in Schank and Colby (eds.), *Computer models of thought and language*, pp. 63-113

Larger scale chunks of knowledge

Robert Abelson, The structure of belief systems, in Schank and Colby (eds.), *Computer models of thought and language*, pp. 287-340

Wallace Chafe, First technical report, contrastive semantics project, Dept. of Linguistics, University of California, Berkeley, 1972.

Eugene Charniak, Toward a model of children's story comprehension, AI TR-266, MIT Artificial Intelligence Laboratory, Cambridge Mass. 1972.

Drew McDermott, Assimilation of new information by a natural language understanding system, MIT AI TR-291, February 1974.

Roger Schank, Identification of conceptualizations underlying natural language, in Schank and Colby (eds.), *Computer models of thought and language*, pp. 187-248

Roger Schank and Kenneth Colby, *Computer models of thought and language*, Freeman, 1973.

Speech Systems

Allen Newell et. al., *Speech understanding systems: final report of a study group*, North Holland/American Elsevier, 1973.

R.D. Reddy, L.D. Erman, R.D. Fennell, and R.B. Neely, The Hearsay speech system: an example of the recognition process, Advance papers, *Third International Joint Conference on Artificial Intelligence*, pp. 185-193

Donald Walker, Speech understanding through syntactic and semantic analysis, Advance papers, *Third International Joint Conference on Artificial Intelligence*, pp. 208-215

W.A. Woods and J. Makhoul, Mechanical inference problems in continuous speech understanding, Advance papers, *Third International Joint Conference on Artificial Intelligence*, pp. 200-207

]|[Lecture 2

SHRDLU: A SYSTEM FOR DIALOG

In this lecture I will describe in some detail a program I wrote at MIT for understanding natural language. There are many things in that program which I would do very differently today, and many new ideas which have been developed since then. Therefore, I do not want to convey the impression that this program represents the last word in how things should be done, but I think it is important to look at a whole program -- to see how things fit together. Many of the newer ideas are still fragments which have not been combined in an integrated way. In this lecture I will use my program to illustrate the range of things which must go into language understanding, and some of the problems of organization in putting together a complete system.

Overview of the system

There are several important basic features of the SHRDLU system. First, it is a dialog system. There is an exchange between the computer and a person. The person can give commands, ask questions and make statements concerning a simple world of discourse, and the system reacts appropriately, carrying out commands, answering questions, and taking in new information.

It is an integrated system which combines knowledge of syntax (the grammar of the language), semantics (the way the language conveys meaning), and reasoning (the ability to make deductions and connect facts in the subject domain).

The system is designed to account for the problems of discourse -- to see each utterance in the context of the more complete discussion. There is a context of things being discussed, a context of the rest of the discussion -- the sentences which have gone before -- and there is a memory of the events which have happened.

The program is based on a belief that success at understanding language depends on a deep knowledge of the subject being discussed. You need more than a dictionary of word meanings and a knowledge of syntactic structure. The system must know the connections of meanings and facts within the domain. This implies that to translate something like a newspaper, we would need to build in all the varied knowledge that a person brings to that task.

In our program, we chose to resolve the dilemma by picking a tiny bit of the world to talk about. Within this mini-world, we can give the computer a deep kind of knowledge, which allows it to do the necessary reasoning. The subject chosen was the world of a toy robot with a simple arm. It can manipulate toy blocks on a table containing simple objects like a box. In the course of a dialog, it can be asked to move the objects around, doing such things as building stacks and putting things into the box. It can be questioned about the current configuration of blocks on the table, about the events which have gone on during the discussion, and to a limited extent about its reasoning. It can be told simple facts which it adds to its store of knowledge for use in later reasoning. The conversation goes on within a dynamic situation, in which the computer is an active participant, doing things which change his toy world, and discussing them. The program displays a simulated robot world on a TV screen, and converses with a human on a teletype. It was not written for any particular use with a real robot, and does not have a model of language based on peculiarities of the robot environment. Rather, it is precisely by limiting the subject matter to such a small area that we can address the general issues of how language is used in situations involving physical objects, events, and a continuing discourse.

The programs can be roughly divided into the three domains mentioned above: There is a syntactic parser which works with a large-scale grammar of English; there is a collection of semantic routines which embody the kind of knowledge needed to interpret the meanings of words and structures; and there is a cognitive deductive system for exploring the consequences of facts, making plans to carry out commands and finding the answers to questions. There is also a comparatively simple set of programs for generating appropriate English responses.

In designing these pieces, the main emphasis was on their interaction. The form in which we want to state a syntactic theory or a type of deduction must take into account the fact that it is only a part of a larger system. One of the most useful organizing principles was the representation of much of the knowledge as procedures. Many other theories of language state their rules in a form which is modelled on the equations of mathematics, or the rules of symbolic logic. These are static rules which do not explicitly describe the process involved in using them, but which are instead manipulated by some sort of uniform deduction procedure. By writing special languages suited to the various types of knowledge (semantic, syntactic, deductive), we are able to preserve the simplicity of these systems, while putting the knowledge in the form of programs. In these programs we can explicitly express the connections between the different parts of the system's knowledge, enriching their possibilities for interaction.

A dialog with SHRDLU

At this point in the lecture, a film was shown demonstrating a dialog with the system. In these notes I have substituted an account of such a dialog, as published in Winograd (1972). The boldface lines in lower case were typed in by a person, and the boldface lines in upper case are the computer's response. Other lines are comments describing some of the features being exhibited.

The dialog was carried out in real time with the response of the "robot" displayed on a CRT screen. Figure 2.1 shows a typcial scene in the robot's world.

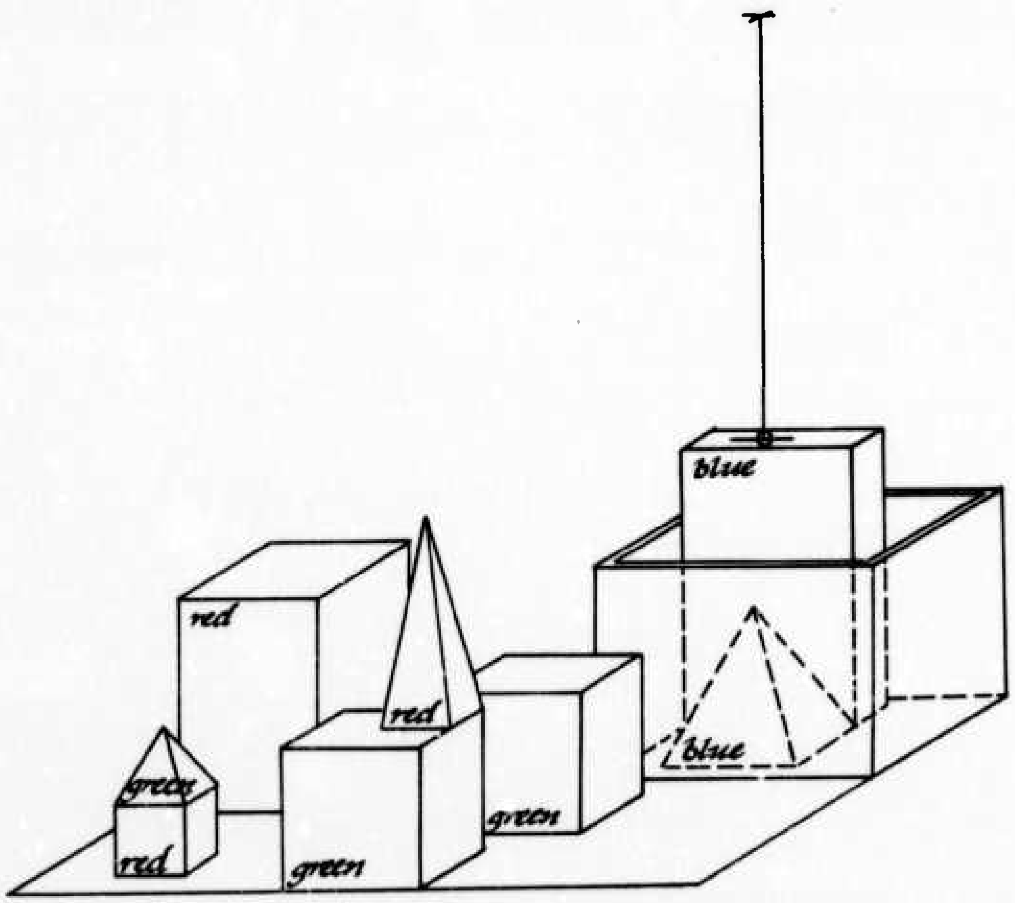

Figure 2.1. The robot's environment.

1. pick up a big red block.

OK.

> The system answers "OK" when it carries out a command. In order to pick up the red block, it had to clear it off by finding a space for the green one and moving the green one away.

2. grasp the pyramid.

I DON'T UNDERSTAND WHICH PYRAMID YOU MEAN.

There are three pyramids in the scene, and the program knows that phrases beginning with "the" are intended to refer to a specific object the speaker has in mind.

3. **find a block which is taller than the one you are holding and put it into the box.**

The system has to figure out what is meant by "one" and "it", as well as using deduction to decide which block it is holding.

BY "IT", I ASSUME YOU MEAN THE BLOCK WHICH IS TALLER THAN THE ONE I AM HOLDING.

There were two possible referents for "it": the one selected, and "the one you are holding". The program has heuristics for choosing between them.

OK.

The hand has to set down the red block to pick up the taller blue one. Figure 2.1 shows the result of the system's manipulations at this point.

4. **what does the box contain?**

THE BLUE PYRAMID AND THE BLUE BLOCK.

5. **what is the pyramid supported by?**

Earlier the system responded to "the pyramid" by saying "I don't understand which pyramid you mean." Now, it assumes that we mean the one just mentioned.

THE BOX.

6. **how many blocks are not in the box?**

Negatives such as "not" and "no" are handled whether they appear in clauses or noun groups (such as "no boxes").

FOUR OF THEM.

7. **is at least one of them narrower than the one which i told you to pick up?**

The phrase "at least one of them" must be analyzed to refer back to the previous question, and "the one which..." must also refer to blocks. Noun groups can have complicated numbers such as "at least one", "more than three", "exactly seven", etc. The memory of past events is used to deduce which block is "the one which I told you to pick up". Notice the complicated syntactic structure in which "the one" is actually the object of a clause embedded two levels deeper. The parser incorporates a grammar which handles this kind of complexity in a general way.

YES, THE RED CUBE.

8. is it supported?

YES, BY THE TABLE.

The system has heuristics about what information is relevant to a person who asks a question, even if he does not request that information specifically.

9. can the table pick up blocks?

NO.

The semantic rules include the knowledge that "picking up" must be done by an animate object, and that the table is not animate.

10. can a pyramid be supported by a block?

YES.

The deductive system finds an actual example, so it knows this is possible.

11. can a pyramid support a pyramid?

I DON'T KNOW.

The question makes sense, but there is no example in the scene. The system has no way of looking into the details of its programs, even though these ultimately define its capabilities.

12. stack up two pyramids.

I CAN'T.

The robot tries and fails.

13. the blue pyramid is mine.

I UNDERSTAND.

All of the information about colors, shapes, sizes and locations of the objects is known to the program initially. Possession is left as an unknown property of objects about which it can be told. It stores this information in its data base.

14. **i own all blocks which are not red, but i don't own anything which supports a pyramid.**

I UNDERSTAND.

The result of understanding these sentences is the creation of new procedures within the robot for deducing whether I own something.

15. **do i own the box?**

NO.

The box supports whatever it contains, including a pyramid.

16. **do i own anything in the box?**

YES, TWO THINGS: THE BLUE BLOCK AND THE BLUE PYRAMID.

17. **will you please stack up both of the red blocks and either a green cube or a pyramid?**

OK.

Logical connectives such as "and" "or" "either" etc. are handled in both the grammar and semantics. A command is executed even if it looks like a question. The command is carried out by the following series of steps: It puts a green cube on the large red block (It chooses the green cube with nothing on it.), then removes the small pyramid from the little red cube, so that it can use that cube to complete the stack as specified, resulting in the situation of Figure 2.2 .

18. **which cube is sitting on the table?**

THE LARGE GREEN ONE WHICH SUPPORTS THE RED PYRAMID.

Ordinarily the system would be satisfied with calling this "a big green cube", but since we asked "which cube" it has to be more specific in naming it. It has procedures for using substitute nouns (such as "one") and pronouns in answers to make the discourse more natural.

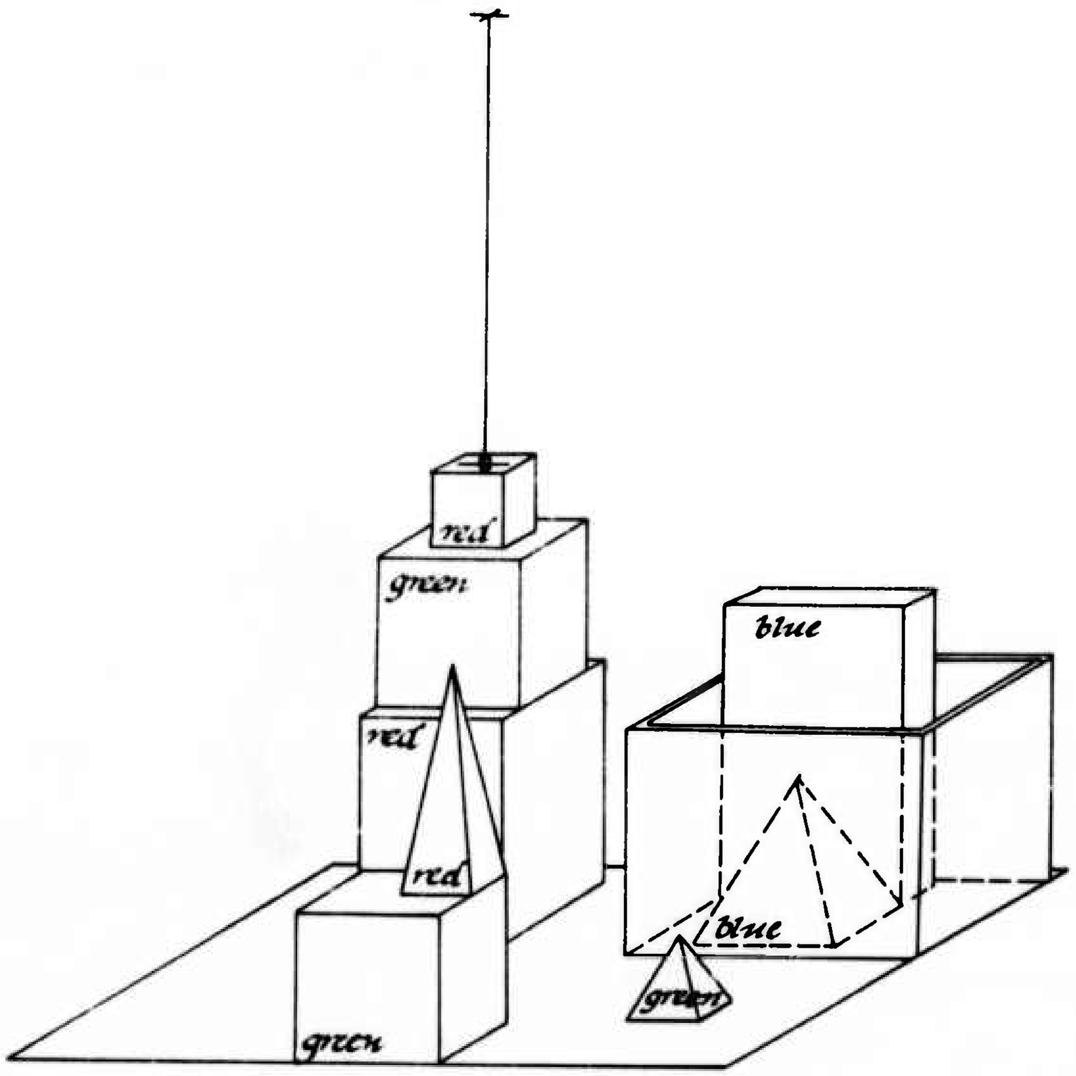

Figure 2.2. The result of building a stack.

19. is there a large block behind a pyramid?

YES, THREE OF THEM: A LARGE RED ONE, A LARGE GREEN CUBE AND THE BLUE ONE.

20. put a small one onto the green cube which supports a pyramid.

> "A small one" must contrast with the person's earlier phrase "a large block", so this means "a small block".

OK.

21. put the littlest pyramid on top of it.

> Words like "littlest" are not in the dictionary but are interpreted from the root forms like "little". The program compares the sizes of all the pyramids to carry out the action.

OK.

22. how many things are on top of green cubes?

I'M NOT SURE WHAT YOU MEAN BY "ON TOP OF" IN THE PHRASE "ON TOP OF GREEN CUBES ".

DO YOU MEAN:
1 - DIRECTLY ON THE SURFACE
2 - ANYWHERE ON TOP OF ?

> In the previous sentence, the phrase "on top of" was interpreted without trouble because the context made it clear what was meant. In this case, the heuristics for disambiguation do not come up with a clearcut decision, so the system has to ask the user for more information. The paraphrases were included with the dictionary definition of "on". A request like this can also occur when the meaning of a pronoun such as "it" or "they" is unclear.

2

THREE OF THEM.

23. had you touched any pyramid before you put the green one on the little cube?

YES, THE GREEN ONE.

> The system deduces from its memory what event is being referred to, and can relate other events to it in time. The analysis includes complex tenses of verb groups, and there are special facilities for keeping track of the various forms of irregular verbs like "have".

24. when did you pick it up?

WHILE I WAS STACKING UP THE RED CUBE, A LARGE RED BLOCK AND A LARGE GREEN CUBE.

> A time is described by naming the main goal which was being worked on.

25. why?

> The system remembers immediately previous sentences to understand questions like this and the following ones.

TO GET RID OF IT.

> By keeping track of selected parts of the original subgoal tree, the system has some understanding of its own motives.

26. why did you do that?

TO CLEAN OFF THE RED CUBE.

> "That" refers to the action mentioned in the system's last answer.

27. why did you clear off that cube?

> Here, "that" means the one just referred to.

TO PUT IT ON A LARGE GREEN CUBE.

28. why did you do that?

TO STACK UP THE RED CUBE , A LARGE RED BLOCK AND A LARGE GREEN CUBE.

29. why did you do that?

BECAUSE YOU ASKED ME TO.

> It remembers what it did, not how the request was worded.

30. how did you do it?

> "It" refers to the action mentioned in the last question, rather than the answer.

BY PUTTING A LARGE RED BLOCK ON THE TABLE ; THEN LETTING GO OF IT ; THEN PUTTING A LARGE GREEN CUBE ON IT ; THEN LETTING GO OF THAT CUBE ; THEN PUTTING THE RED CUBE ON THAT CUBE ; THEN LETTING GO OF THAT CUBE.

> This is a description of the top-level subgoals used to achieve the goal.

31. how many objects did you touch while you were doing it?

> Time references can refer to a time span, using words like "while".

FOUR OF THEM.

32. what did the red cube support before you started to clean it off?

THE GREEN PYRAMID.

> By remembering what has been moved, it is possible to deduce things about earlier states of the scene. We can use complex nestings of clauses like "before you started to clean it off".

33. there were five blocks to the left of the box then.

> "Then" must refer to the time mentioned in the last sentence.

NO, ONLY FOUR OF THEM: THE RED CUBE , TWO LARGE GREEN CUBES AND A LARGE RED BLOCK.

> Earlier in the dialog, information about "owning" was accepted at face value. But the system knows that it has complete information about the basic properties of objects, including location. It checks any statements about these to see whether it agrees or disagrees.

34. put the blue pyramid on the block in the box.

> This is syntactically ambiguous, but is understood without ambiguity since the parsing corresponding to "put the blue pyramid which is on the block into the box." does not make sense in the current scene, while "put the blue pyramid onto the block which is in the box." does.

OK.

35. is there anything which is bigger than every pyramid but is not as wide as the thing that supports it?

> The system understands quantifiers such as "every", "any", "some", etc., and structures such as "bigger than", "as wide as", etc. Here "it" refers to a noun group within the same sentence but two levels up in the parsing tree.

YES, THE BLUE BLOCK.

36. does a steeple

SORRY, I DON'T KNOW THE WORD "STEEPLE".

> The system interrupts immediately when it sees a word it doesn't know.

37. a "steeple" is a stack which contains two green cubes and a pyramid.

> The quote marks indicate that we are defining a new word. The definition is added to the dictionary in a form compatible with all the words it already knows.

I UNDERSTAND.

38. are there any steeples now?

NO.

> The new definition is used as a description to recognize steeples.

39. build one.

OK.

> The description given in the definition is enough to deduce a plan for building new objects as well as recognizing them. The word "one" means "a steeple", and the plan is carried out. First, one of the large green cubes is cleared off by removing the red cube and two pyramids. Then the two green cubes and the tall red pyramid are stacked up to fit the description.

40. call the biggest block "superblock".

OK.

> We can also give objects names.

41. have you picked up superblock since we began?

YES.

42. why did you drop it?

BY "IT", I ASSUME YOU MEAN THE SUPERBLOCK.

I CAN'T EXPLAIN A NON-EXISTENT EVENT.

> There is no record in memory of any block being dropped. The answer routines have special checks for this type of "loaded question" which presupposes things which aren't true.

43. is there anything to the right of the red pyramid?

YES, FOUR THINGS: THE BOX , THE RED CUBE, THE BLUE PYRAMID AND THE BLOCK NAMED SUPERBLOCK

Once an object has a name, the name is used to help identify it in responses.

44. thank you.

YOU'RE WELCOME!

The grammar is written as a procedure, which gives it the capability to carry out special sorts of parsing for idioms.

The components of the system

In this section I will describe in some detail the various parts of the language understanding program and its operation. This type of analysis is important for understanding any computer model, since the surface appearance of its output can be deceptive. A program may succeed by carefully chosing the problem it will attack, so that some simple special-purpose solution will work. In the previous lecture I described ELIZA and STUDENT as examples of programs which give impressive performances due to a severe and careful restriction of the kind of understanding they try to achieve. If a model is to be of broader significance, it must be designed to cover a large range of the things we mean when we talk of understanding. The principles should derive from an attempt to deal with the basic cognitive structures. In the rest of the lecture, I would like to give some feeling for the set of ideas used in representing knowledge in this system.

Reasoning

First, let us look at its knowledge of the blocks world. The program makes use of a detailed world model, describing both the current state of the situation and its knowledge of procedures for changing that state and making deductions about it. This model is not in spatial or analog terms, but is a symbolic description, abstracting those aspects of the world which are relevant to the operations used in working with it and discussing it. First there is a data base of simple facts like those shown in Figure 2.3 , describing what is true at any particular time. There we see, for example, that B1 is a block, B1 is red, B2 supports B3, blue is a color, EVENT27 caused EVENT29, etc. The notation involves simply indicating relationships between objects by listing the name of the relation (such as IS or SUPPORT) followed by the things being related. These include both concepts (like BLOCK or BLUE) and proper names of individual objects and events (indicated by the use of numbers, like B1 and TABLE2).

The symbols used in these expressions represent the concepts (or conceptual categories) which form the vocabulary of the language user's cognitive model. A concept corresponds vaguely to what we might call a single meaning of a word, but the connection is more complex. Underlying the organization is a belief that meanings cannot be reduced to any set of pure "elements" or components from which everything else is built. Rather, a person categorizes his experience along lines which are relevant to the thought processes he will use, and his categorization is generally neither consistent, parsimonious, nor complete. A person may categorize a set of objects in his experience into, for example "chair", "stool", "bench", etc. If pushed, he cannot give an exact definition for any of these, and in naming some objects he will not be certain how to make the choice between them. This is even clearer if we consider words like "truth", "virtue", and "democracy". The meaning of any concept depends on its interconnection with all of the other concepts in the model.

```
(IS B1 BLOCK)
(IS B2 PYRAMID)
(AT B1 (LOCATION 100 100 0))
(SUPPORT B1 B2)
(CLEARTOP B2)
(MANIPULABLE B1)
(CONTAIN BOX1 B4)
(COLOR-OF B1 RED)
(SHAPE-OF B2 POINTED)
(IS BLUE COLOR)
(CAUSE EVENT23 EVENT25)
```

Figure 2.3. Some assertions about the BLOCKS world.

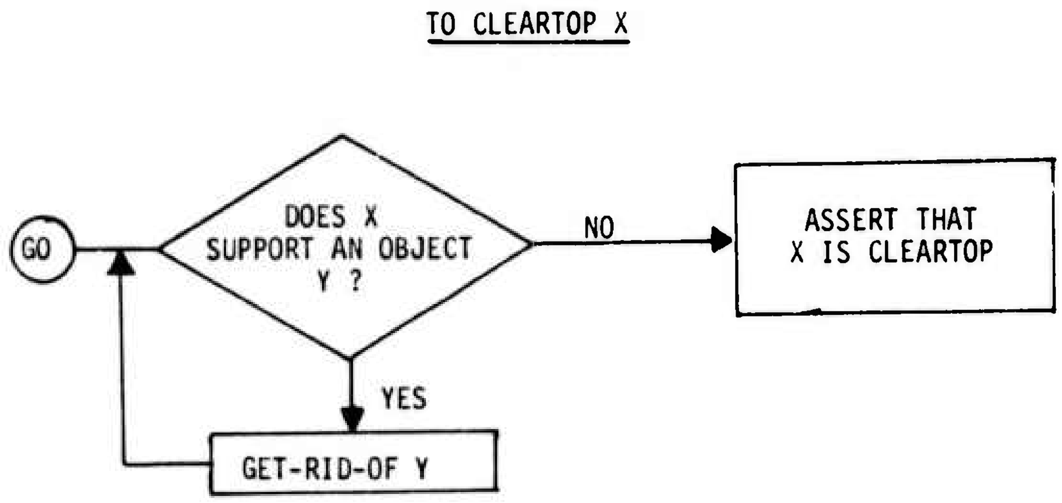

Figure 2.4. Flow chart for the concept CLEARTOP.

Most formal approaches to language have avoided this characterization of meaning even though it seems close to our intuitions about how language is used. This is because the usual techniques of logic and mathematics are not easily applicable to such "holistic" models. With such a complex notion of "concept", we are unable to prove anything about meaning in the usual mathematical notion of proof. One important aspect of computational approaches to modelling cognitive processes is their ability to deal with this sort of formalism. Rather than trying to prove things about meaning we can design procedures which can operate with the model, and simulate the processes involved in human use of meaning. The justification for the formalism is the degree to which it succeeds in providing a model of understanding.

What is important then, is the part of the system's knowledge which involves the interconnections among the concepts. In our model, these are in the form of procedures written in the Micro-planner programming language. For example, the concept CLEARTOP (which might be expressed in English by a phrase like "clear off") can be described by the procedure diagrammed in Figure 2.4 .

This flow-chart indicates that to clear off an object X, we start by checking to see whether X supports an object Y. If so, we GET-RID-OF Y, and go check again. When X does not support any object, we can assert that it is CLEARTOP. In this operational definition, we call on other concepts like GET-RID-OF and SUPPORT. Each of these in turn is a procedure, involving other concepts like PICKUP and GRASP. This representation is oriented to a model of deduction in which we try to satisfy some goal by setting up successive subgoals, which must be achieved in order to eventually satisfy the main goal.

Looking at the flow chart for GRASP in Figure 2.5 , we can see the steps the program would take if asked to grasp an object B1 while holding a different object B2. It would be called by setting up a goal of the form (GRASP B1), so when the GRASP program ran, X would represent the object B1. First it checks to see whether B1 is a manipulable object, since if not the effort must fail. Next it sees if it is already grasping B1, since this would satisfy the goal immediately. Then, it checks to see if it is holding an object other than B1, and if so tries to GET-RID-OF it. The program for GET-RID-OF tries to put the designated object on the table by calling a program for PUTON, which in turn looks for an empty location and calls PUT. PUT deduces where the hand must be moved and calls MOVEHAND. If we look at the set of currently active goals at this point, we get the stack in Figure 2.6

Notice that this subgoal structure provides the basis for asking "why" questions, as in sentences 25 through 29 of the dialog in Section 2. If asked "Why did you put B2 on the table?", the program would look to the goal that called PUTON, and say "To get rid of it." If asked "Why did you get rid of it?" it would go up one more step to get "To grasp B1" (Actually, it would generate an English phrase describing the object B1 in terms of its shape, size, and color.) "How" questions are answered by looking at the set of subgoals called directly in achieving a goal, and generating descriptions of the actions involved.

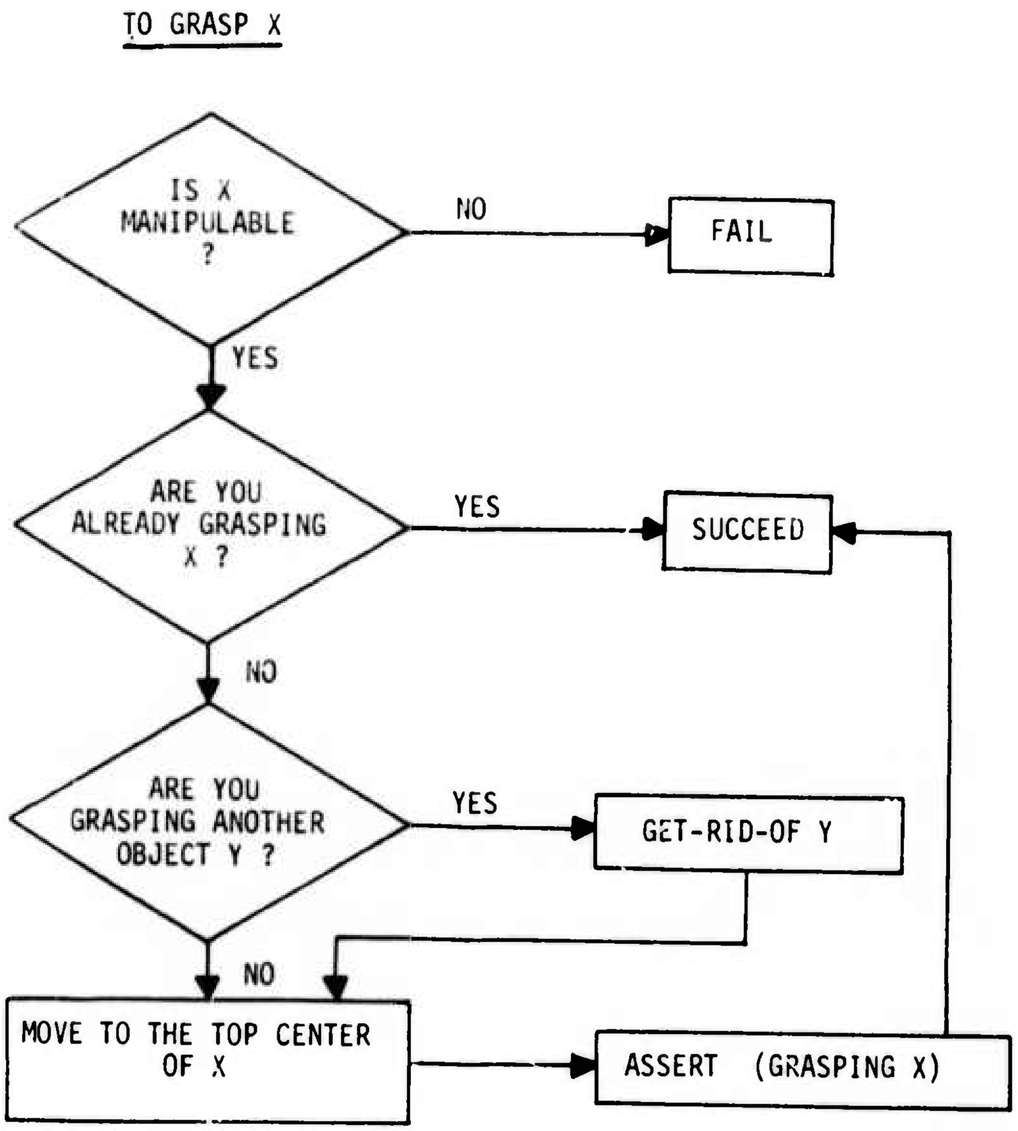
TO GRASP X
IS X MANIPULABLE ?
NO
FAIL
YES
ARE YOU ALREADY GRASPING X ?
YES
SUCCEED
NO
ARE YOU GRASPING ANOTHER OBJECT Y ?
YES
GET-RID-OF Y
NO
MOVE TO THE TOP CENTER OF X
ASSERT (GRASPING X)

Figure 2.5. The procedure for GRASP.

```
(GRASP B1)
     (GET-RID-OF B2)
          (PUTON B2 TABLE1)
               (PUT B2 (453 201 0))
                    (MOVEHAND (553 301 100))
```

Figure 2.6. Stack of goals for grasping object B1.

These examples illustrate the use of procedural descriptions of concepts for carrying out commands, but they can also be applied to other aspects of language, such as questions and statements. One of the basic viewpoints underlying the model is that all language use can be thought of as a way of activating procedures within the hearer. We can think of any utterance as a program, which indirectly causes a set of operations to be carried out within his cognitive system. This "program writing" is indirect in the sense that we are dealing with an intelligent interpreter, who may take a set of actions which are quite different from those the speaker intended. The exact form is determined by his knowledge of the world, his expectations about the person talking to him, his goals, etc. In this program we have a simple model of this process of interpretation as it takes place in the robot. Each input sentence is converted to a set of instructions in PLANNER, which is then executed to achieve the desired effect. In some cases the procedure invoked involves direct physical actions like those above. In others, it may be a search for some sort of information (perhaps to answer a question), while in still others it is a procedure which stores away a new piece of knowledge, or modifies the knowledge it already has.

Let us look at what the system would do with a simple description like "a red cube which supports a pyramid." The description will use concepts like BLOCK, RED, PYRAMID, and EQUIDIMENSIONAL, which are parts of the system's underlying categorization of the world. The result can be represented in a flow chart like that of Figure 2.7 . Note that this is a program for finding an object fitting the description. It would then be incorporated into a command for doing something with the object, a question asking something about it, or, if it appeared in a statement, it would become part of the program which was generated to represent the meaning for later use. Note that this bit of program could also be used as a test to see whether an object fit the description, if the first FIND instruction were told in advance to look only at that particular object.

At first glance, it seems like there is too much structure in this program. We don't like to think of the meaning of a simple phrase as explicitly containing loops, conditional tests, and other programming details. The solution is in providing an internal language which contains the appropriate looping and checking as its primitives, and in which the representation of the process is as simple as the description. PLANNER provides these primitives in our system. The program described in Figure 2.7 would be written in PLANNER looking something like Figure 2.8 . The loops of the flow chart are implicit in PLANNER'S backtrack control structure.

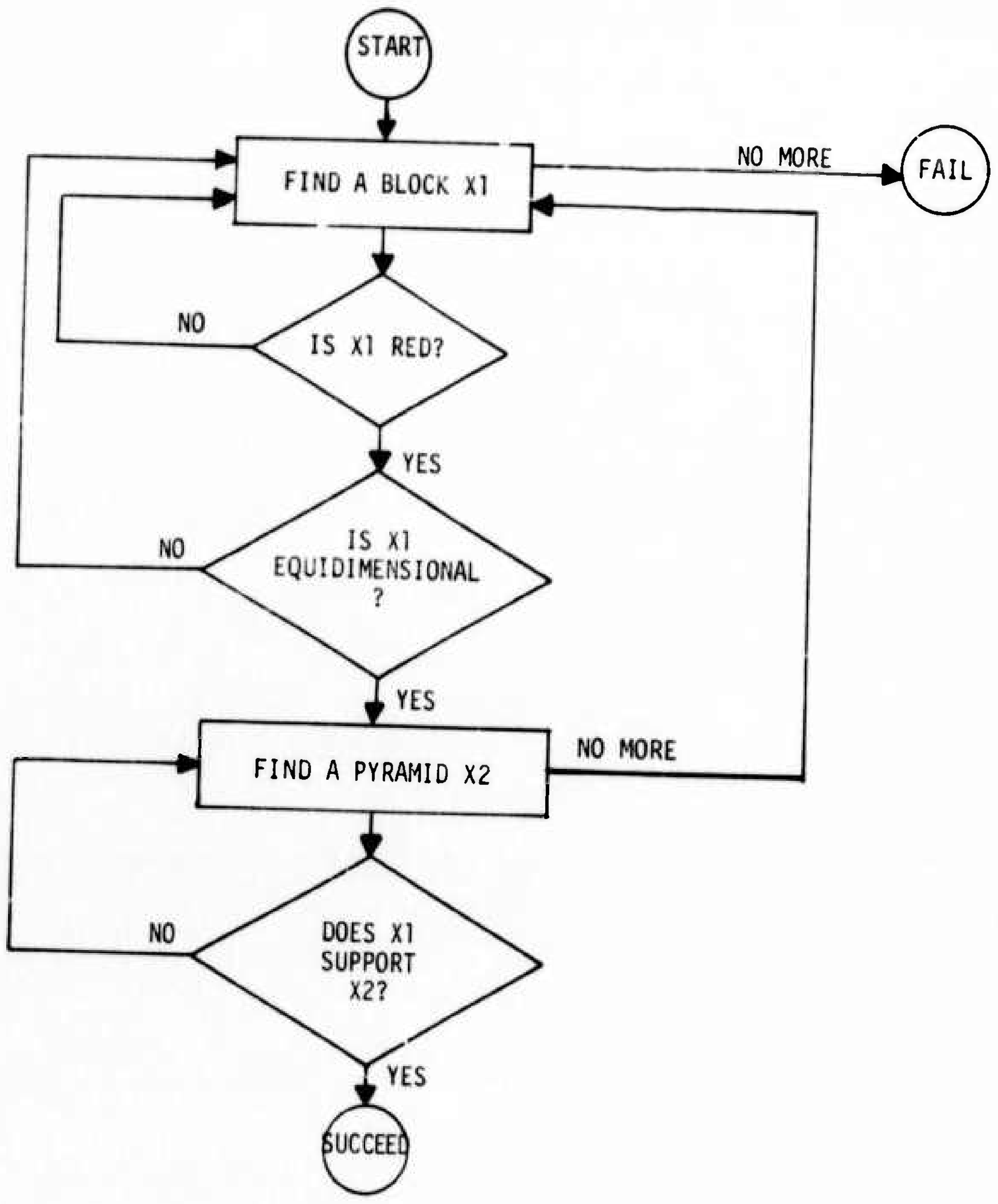

Figure 2.7. Program to find "a red cube which supports a pyramid."

```
(GOAL (IS ?X1 BLOCK))
(GOAL (COLOR-OF ?X1 RED))
(GOAL (EQUIDIMENSIONAL ?X1))
(GOAL (IS ?X2 PYRAMID))
(GOAL (SUPPORT ?X1 ?X2))
```

Figure 2.8. Micro-planner code for "a red cube which supports a pyramid."

The description is evaluated by proceeding down the list until some goal fails, at which time the system backs up automatically to the last point where a decision was made, trying a different possibility. A decision can be made whenever a new object name is assigned to a variable. Using other primitives of PLANNER, such as NOT and FIND (which looks for a given number of objects fitting a description), we can write procedural representations for a wide range of descriptions.

Semantics

Once we have decided how the system will represent meanings internally, we must deal with the way in which it creates a program given an English input. There must be ways to interpret the meanings of individual words and the syntactic structures in which they occur. First, let us look at how we can define simple words like "cube", and "contain". The definitions in Figure 2.9 are completely equivalent to those used in the program.

```
(CUBE
    ((NOUN (OBJECT
               ((MANIPULABLE RECTANGULAR)
                       ((IS ? BLOCK)
                       (EQUIDIMENSIONAL ?)))))))
(CONTAIN
    ((VERB ((TRANSITIVE (RELATION
               (((CONTAINER)) ((PHYSICAL-OBJECT))
                       (CONTAIN 1 2))
               (((CONSTRUCT)) ((PHYSICAL-OBJECT))
                       (PART-OF 2 1)))))))))
```

Figure 2.9. Dictionary definitions for "cube" and "contain".

Their interpretation is straightforward. The first says that a cube is an object which is RECTANGULAR and MANIPULABLE, and can be recognized by the fact that it is a BLOCK, and is EQUIDIMENSIONAL. The first part of this definition is based on the use of semantic markers, and is for efficiency in choosing interpretations. By making a rough categorization of the objects in the model, the system can make quick checks to see whether certain combinations are ruled out by simple tests like "this meaning of the adjective applies only to words which represent physical objects." Chomsky's famous sentence "Colorless green ideas sleep furiously." would be eliminated easily by such markers.

The system uses this information, for example, in answering question 9 in the dialog, "Can the table pick up blocks?", as "pick up" demands a subject which is ANIMATE, while "table" has the marker INANIMATE. These markers are a useful but rough approximation to the actual deductions a person uses in such cases.

The definition for "contain" shows how they might be used to choose between possible word meanings. If applied to a CONTAINER and a PHYSICAL-OBJECT, as in "The box contains three pyramids.", the word implies the usual relationship we mean by CONTAIN. If instead, it applies to a CONSTRUCT (like "stack", "pile", or "row")and an object, the meaning is different. "The stack contains a cube." really means that a cube is PART of the stack, and the system will choose this meaning by noting that CONSTRUCT is one of the semantic markers of the word "stack" when it applies the definition.

One important aspect of these definitions is that although they look like static rule statements, they are actually calls to programs (OBJECT and RELATION) which do the appropriate checks and build the semantic structures. Once we get away from the simplest words, these programs need to be more flexible in what they look at. For example, in the robot world, the phrase "pick up" has different meanings depending on whether it refers to a single object or several. In sentence 1, the system interprets "Pick up the big red block." by grasping it and raising the hand. If we said "Pick up all of your toys." it would interpret "pick up" as meaning "put away", and would pack them all into the box. The program for checking to see whether the object is singular or plural is simple, and any semantic system must have the flexibility to incorporate such things in the word definitions. We do this by having the definition of every word be a program which is called at an appropriate point in the analysis, and which can do arbitrary computations involving the sentence and the present physical situation.

This flexibility is even more important once we get beyond simple words. In defining words like "the", or "of", or the "one" in "Pick up a green one." we can hardly make a simple list of properties and descriptors as in Figure 2.9 . The presence of "one" in a noun group must trigger a program which looks into the previous discourse to see what objects have been mentioned, and can apply various rules and heuristics to determine the appropriate reference. For example it must know that in the phrase "a big red block and a little one," we are referring to "a little red block," not "a little big red block" or simply "a little block." This sort of knowledge is part of a semantic procedure attached to the word "one" in the dictionary.

Words like "the" are more complex. When we use a definite article like "the" or "that" in English, we have in mind a particular object or objects which we expect the hearer to know about. I can talk about "the moon" since there is only one moon we usually talk about. In the context of this article, I can talk about "the dialog", and the reader will understand from the context which dialog I mean. If I am beginning a conversation, I will say "Yesterday I met a strange man." even though I have a particular man in mind, since saying "Yesterday I met the strange man." would imply that the hearer already knows of him. In other cases, "the" is used to convey the information that the object being referred to is unique. If I write "The reason I wrote this paper was...", it implies that there was a single reason, while "A reason I wrote this paper was..." implies that there were others. In the case of generic statements, "the" may be used to refer to a whole class, as in "The albatross is a strange bird." This is a quite different use from the single referent of "The albatross just ate your lunch."

A model of language use must be able to account for the role this type of knowledge plays in understanding. In the procecural model, it is a part of the process of interpretation for the structure in which the relevant word is embedded. The different possibilities for the meaning of "the" are procedures which check various facts about the context, then prescribe actions such as "Look for a unique object in the data base which fits this description." or "Assert that the object being described is unique as far as the speaker is concerned." The program incorporates a variety of heuristics for deciding what part of the context is relevant. For example, it keeps track of when in the dialog something has been mentioned. In sentence 2 of the dialog, "Grasp the pyramid." is rejected since there is no particular pyramid which the system can see as distinguished. However, in sentence 5 it accepts the question "What is the pyramid supported by?" since in the answer to sentence 4 it mentioned a particular pyramid.

This type of knowledge plays a large part in understanding the things that hold a discourse together, such as pronouns, adverbs like "then" and "there", substitute nouns such as "one", phrases beginning with "that", and ellipsis. The system is structured in such a way that the heuristics for each can be expressed as a procedure in a straightforward way.

Syntax

In describing the process of semantic interpretation, we stated that part of the relevant input was the syntactic structure of the sentence. In order to provide this, the program contains a parser and a fairly comprehensive grammar of English. The approach to syntax is based on a belief that the form of syntactic analysis must be usable by a realistic semantic system, and the emphasis of the resulting grammar differs in several ways from traditional transformational approaches.

First, it is organized around looking for syntactic units which play a primary role in determining meaning. A sentence such as "Every musicial likes long romantic sonatas." will be parsed to generate the structure shown in Figure 2.10 . The noun groups (NG) correspond to descriptions of objects, while the clause is a description of a relation or event. The semantic programs are organized into groups of procedures, each of which is used for interpreting a certain type of unit.

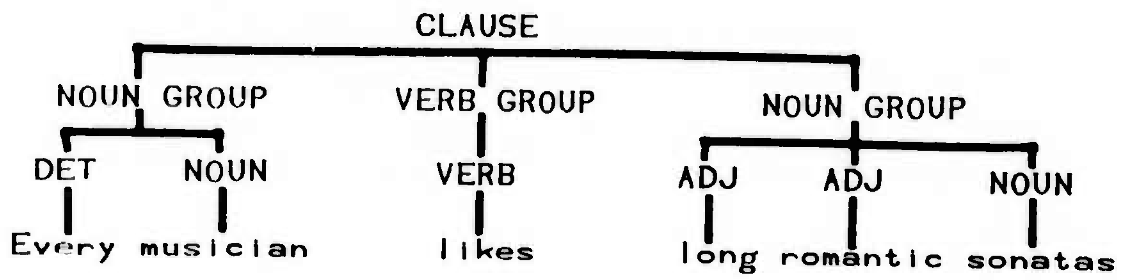

Figure 2.10. Parse tree for a simple sentence.

For each unit, there is a syntactic program (written in a language called PROGRAMMAR, especially designed for the purpose) which operates on the input string to see whether it could represent a unit of that type. In doing this, it will call on other such syntactic programs (and possibly on itself recursively). It embodies a description of the possible orderings of words and other units. for example, the scheme for a noun group, as shown in Figure 2.11 .

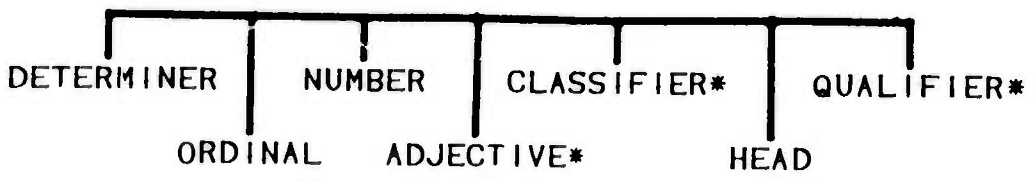

Figure 2.11. Structure of a noun group.

The presence of an asterisk after a symbol means that that function can be filled more than once. The figure shows that we may have a determiner (such as "the") followed by an ordinal (such as "first"), then a number ("three") followed by one or more adjectives ("big," "red") followed by one or more nouns being used as classifiers ("fire hydrant") followed by a noun ("covers") followed by qualifying phrases which are preposition groups or clauses ("without handles" "which you can find"). Of course many of the elements are optional, and there are restriction relations between the various possibilities. If we choose an indefinite determiner such as "a", we cannot have an ordinal and number, as in the illegal string "a first three big red fire hydrant covers without handles you can find." The grammar must be able to express these rules in a way which is not simply an ad hoc set of statements. Our grammar takes advantage of some of the ideas of Systemic Grammar (Halliday, 1971).

Systemic theory views a syntactic structure as being made up of units, each of which can be characterized in terms of the *features* describing its form and the *functions* it fills in a larger structure or discourse. In the sentence in Figure 2.10 the noun group "every musician" can be described as exhibiting features such as QUANTIFIED, UNIVERAL, SINGULAR, etc. It serves the function SUBJECT in the clause of which it is a part, as well as various discourse functions, such as THEME. It in turn is made up of other units -- the individual words -- which fill functions in the noun group, such as DETERMINER and HEAD. A grammar must include a specification of the possible features a unit can have, and the relation of these to both the functions it can play, and the functions and constituents it controls. These features are not haphazard bits of information we might choose to notice about units, but form a highly structured system (hence the name Systemic Grammar). As an example, we can look at a few of the features for the CLAUSE in Figure 2.12 .

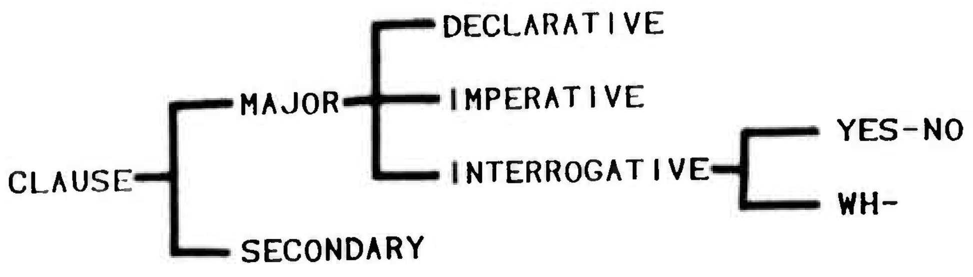

Figure 2.12. Simple system network for the clause.

The vertical lines represent sets from which a single feature must be selected, while horizontal lines indicate logical dependency. Thus, we must first choose whether the clause is MAJOR -- which corresponds to the function of serving as an independent sentence -- or SECONDARY, which corresponds to the various functions a clause can serve as a constituent of another unit (for example as a QUALIFIER in the noun group "the ball *which is on the table*.").

If a clause is MAJOR, it is either DECLARATIVE ("She went.") IMPERATIVE ("Go.") or INTERROGATIVE ("Did she go?"). If it is INTERROGATIVE, there is a further choice between YES-NO ("Did she go?") and WH- ("Where did she go?").

It is important to note that these features are syntactic, not semantic. They do not represent the use of a sentence as a question, statement, or command, but are rather a characterization of its internal structure -- which words follow in what order. A DECLARATIVE can be used as a question by giving it a rising intonation, or even as a command, as in "You're going to give that to me." spoken in an appropriate tone. A question may be used as a polite form of a command, as in "Can you give me a match?", and so on. Any language understander must know the conventions of the language for interpreting such utterances in addition to its simpler forms of syntactic knowledge. To do this, it must have a way to state things like "If something is syntactically a question but involves an event which the hearer could cause in the immediate future, it may be intended as a request." Syntactic features are therefore basic to the description of the semantic rules. The actual features in a comprehensive grammar are interrelated in a more complex way than the simple example of Figure 2.12 , but the basic ideas of logical dependency are the same.

Above we stated that there is a choice between certain features, and that depending on the selection made from one set, we must then choose between certain others. In doing this we are not postulating a psychological model for the order of making choices. The networks are an abstract characterization of the possibilities, and form only a part of a grammar. In addition we need realization and interpretation rules. Realization rules describe how a given set of choices would be expressed in the form of surface syntactic structures, while interpretation rules describe how a string of words is analyzed to find its constituents and their features.

Our grammar is an interpretation grammar for accepting grammatical sentences. It differs from more usual grammars by being written explicitly in the form of a program. Ordinarily, grammars are stated in the form of rules, which are applied in the framework of a special interpretation process. This may be very complex in some cases (such as transformational grammars) with separate phases, special "traffic rules" for applying the other rules in the right order, cycles of application, and other sorts of constraints. In our system, the sequence of the actions is represented explicitly in the set of rules. The process of understanding an utterance is basic to the organization of the grammar.

In saying that grammars are programs, it is important to separate the procedural aspect from the details usually associated with programming. If we say to a linguist "Here is a grammar of English," he can rightfully object if it begins "Take the contents of location 177 and put them into register 2, adding the index..." The formalisation of the syntax should involve only those operations and concepts which are relevant to linguistic analysis, and should not be burdened with paraphernalia needed for programming details. Our model is based on the belief that the basic ideas of programming such as procedure and subprocedure, iteration, recursion, etc. are central to all cognitive processes, and in particular to the theory of language. What is needed is a formalism for describing syntactic processes. Our grammar is written in a language which was designed specifically for the purpose. It is a system built in LISP, called PROGRAMMAR, and its primitive operations are those involving the building of syntactic structures, and the generation of systemic descriptions of their parts. The set of typical grammar rules shown in Figure 2.13 would be expressed in PROGRAMMAR by the program diagrammed in Figure 2.14 . For such a simplified bit of grammar, there isn't much difference between the two formulations, except that the PROGRAMMAR representation is more explicit in describing the flow of control.

```
  S → NP VP
 NP → DETERMINER NOUN
 VP → VERB/TRANSITIVE NP
 VP → VERB/INTRANSITIVE
```

Figure 2.13. Simple grammar rules.

When we try to deal with more complex parts of syntax, the ability to specify procedures becomes more important. For example the word "and" can be associated with a program which can be diagrammed as shown in Figure 2.15 .

Of course the use of conjunctions is more complex than this, and the actual program must take into account such things as lists and branched structures, and the problems of backing up if a wrong possibility has been tried. But the basic operation of "look for another one like the one you just found" seems both practical and intuitively plausible as a description of how conjunction works. The ability to write the rules as procedures leaves us the flexibility to extend and refine it.

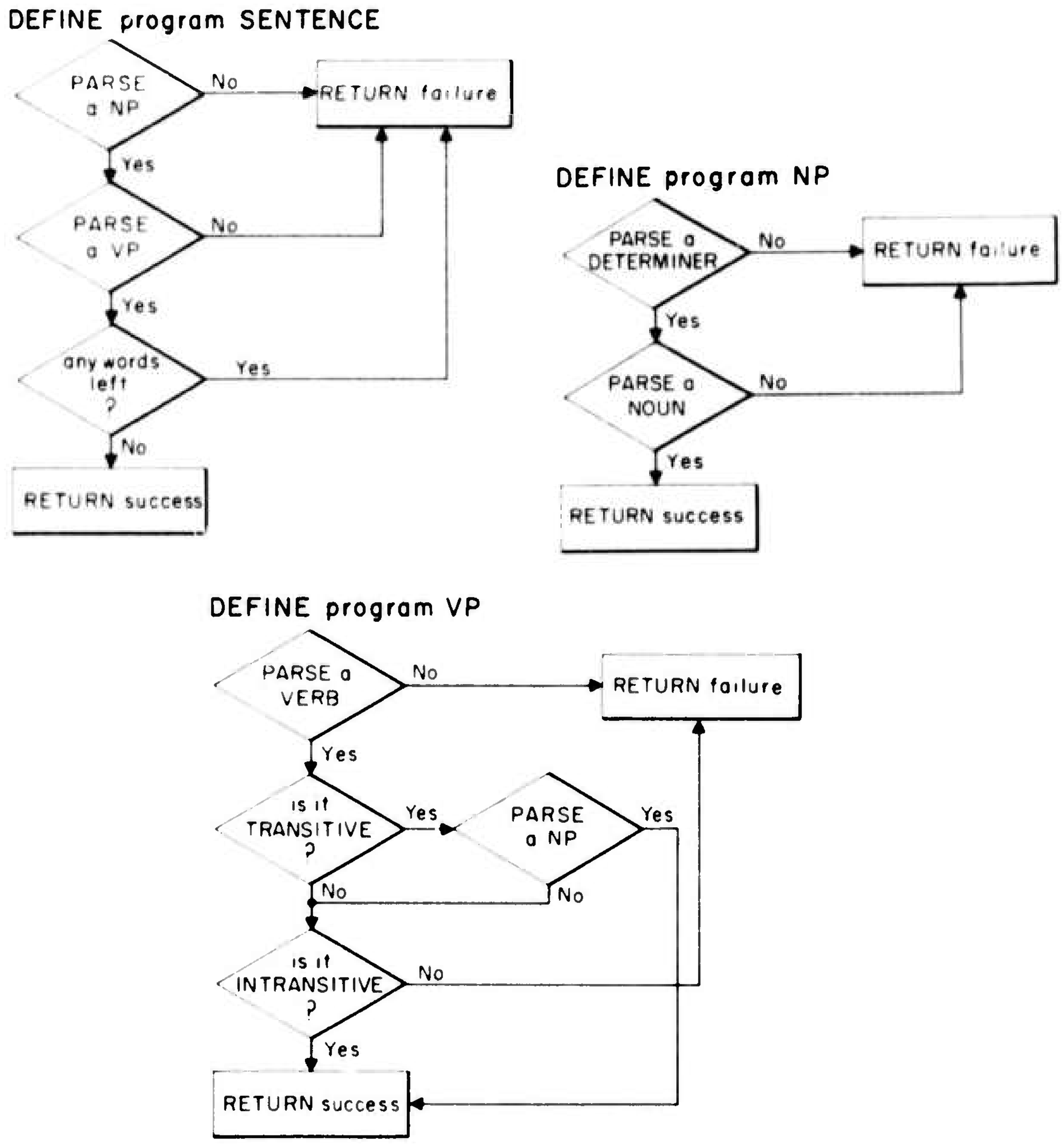

Figure 2.14. Simple procedural grammar.

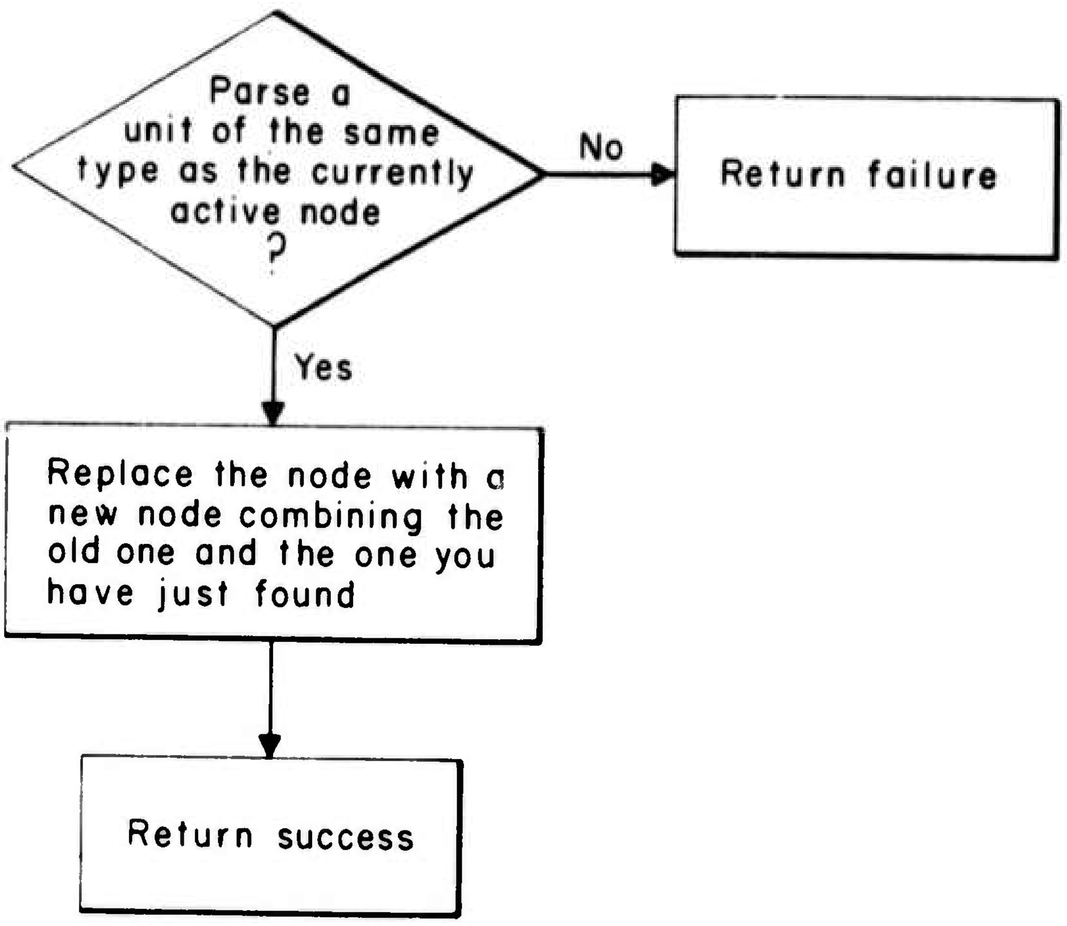
Parse a unit of the same type as the currently active node ?
No
Return failure
Yes
Replace the node with a new node combining the old one and the one you have just found
Return success

Figure 2.15. Program for the syntax of "and".

Operation of the system

So far, we have described how three different types of knowledge are represented and used. There is the data base of assertions and PLANNER procedures which represent the knowledge of the physical world; there are semantic analysis programs which know about such problems as reference; and there is a grammar which determines the syntactic structure. The most important element, however, is the interaction between these components. Language cannot be reduced into separate areas such as "syntax, semantics, and pragmatics" in hopes that by understanding each of them separately, we have understood the whole. The key to the function of language as a means of communication is in the way these areas interact.

Our program does not operate by first parsing a sentence, then doing semantic analysis, then using deduction to produce a response. The three go on concurrently throughout the understanding of a sentence. As soon as a piece of syntactic structure begins to take shape, a semantic program is called to see whether it might make sense, and the resultant answer can direct the parsing. In deciding whether it makes sense, the semantic routine may call deductive processes and ask questions about the real world. As an example, in sentence 36 of the dialog ("Put the blue pyramid on the block in the box.") the parser first comes up with "the blue pyramid on the block" as a candidate for a noun group. At this point, semantic analysis is begun, and since "the" is definite, a check is made in the data base for the object being referred to. When no such object is found, the parsing is redirected to find the noun group "the blue pyramid". It will then go on to find "on the block in the box" as a single phrase indicating a location. In other cases, the system of semantic markers may reject a possible interpretation on the basis of conflicting category information. Thus, there is a continuing interplay between the different sorts of analysis, with the results of one affecting the others.

The procedure as a whole operates in a left to right direction through the sentence. It does not carry along multiple possibilities for the syntactic analysis, but instead has ways of going back and doing something different if it runs into trouble. It does not use the general backup mechanism of PLANNER, but decides what to do on the basis of exactly what sort of problem arose. In the sentences like those of the dialog, very little backup is ever used, since the combination of syntactic and semantic information usually guides the parser down profitable paths.

Areas of inadequacy

Looking into the specific capabilities of the system, we can find many places where the details seem inadequate, or whole areas are missing. It does not attempt to handle hypothetical or counterfactual statements, it only accepts a limited range of declarative information, it cannot talk about verbal acts, the treatment of "the" is not as general as the description above, and so on. However, these deficiencies seem to be more a matter of what has been tackled so far, rather than calling into question the underlying model. Looking deeper, we can find two basic ways in which it seems an inadequate model of human language use. The first involves the way in which the process is directed, and the second is concerned with the interaction between the context of the conversation and the understanding of its content.

The understanding process

We can think of a program for understanding a sentence as involving two kinds of operations -- coming up with possible interpretations, and choosing between them. Of course, these are not two separate components from a psychological standpoint, but in the organization of computer programs, the work is divided up.

In our program, the syntactic analysis is in charge of coming up with possibilities. The basic operation involves finding a syntactically acceptable phrase, then doing semantic interpretation on it to decide whether to continue along that line of parsing. Other programs such as those of Schank and Quillian use the semantic information contained in the definitions of the words to provide an initial set of possibilities, then use syntactic information in a secondary way to check whether the hypothesized underlying semantic structure is in accord with the arrangement of the words.

In looking at human language use, it seems clear that no single approach is really correct. On the one hand, people are able to interpret utterances which are not syntactically well formed, and can even assign meanings to collections of words without use of syntax. The list "skid, crash, hospital" presents a certain image, even though two of the words are both nouns and verbs and there are no explicit syntactic connections. It is therefore wrong to insist that some sort of complete parsing is a prerequisite for semantic analysis.

On the other hand, people are able to interpret sentences syntactically even when they do not know the meanings of the individual words. Most of our vocabulary (beyond a certain age) is learned by hearing sentences in which unfamiliar words appear in syntactically well-defined positions. We process the sentence without knowing any category information for the words, and in fact use the results of that processing to discover the semantic meaning. In addition, much of our normal conversation is made up of sentences like "Then the other one did the same thing to it." in which the words taken individually do not provide the clues which would enable us to determine the conceptual structure without a complete syntactic analysis.

What really seems to be going on is a coordinated process in which a variety of syntactic and semantic information can be relevant, and in which the hearer takes advantage of whatever is more useful in understanding a given part of a sentence. Our system models this coordination in its order of doing things, by carrying on all of the different levels of analysis concurrently. However it does not model it in the control structure.

Much remains to be done in understanding how to write computer programs in which a number of concurrent processes are working in a coordinated fashion without being under the primary hierarchical control of one of them. A language model which is able to really implement the sort of "heterarchy" found in biological systems (like the coordination between different systems of an organism) will be much closer to a valid psychological theory.

We might imagine a system which operated like a person putting together a jigsaw puzzle. The shape of the pieces might correspond to syntax -- there are rules for how the different shapes fit together, and some pieces can be assembled without regard to what appears on them. Most of the time, though it is much easier to use information like "This piece is red, so if I could find a piece with a red tab on it, it might fit". The search for the next piece is based on color rather than shape. We might view things like color and texture as a kind of simple picture semantics which indicates what sorts of elements can fit with what others. This often serves as the basic way of deciding what pieces to try, while the exact grammar of the shape is used to check each possibility and see if it really fits. Finally, there is a more sophisticated pragmatics or reasoning based on knowledge of pictures. If a picture of an elephant is emerging, it might be useful to look for something with the color and texture of an elephant tail, and then use its further color and shape information to guide the process.

This jigsaw style of organization makes flexible use of whatever information is most helpful at a given moment, using other sources of information as a check on that. Systems like the Hearsay speech system are beginning to explore this style of organization, but most current language systems are driven by a single primary aspect. SHRDLU can be viewed as attempting to fit pieces on the basis of shape, then checking both the colors and patterns to see whether they fit or something else should be tried. Schank's system uses the general color features to make proposals, checking the exact shape afterwards.

Natural communication

The second basic area of shortcoming is in not dealing with all the implications of viewing language as a process of communication between two intelligent people. A human language user is always engaged in a process of trying to understand the world around him, including the person he is talking to. He is actively constructing models and hypotheses, and he makes use of them in the process of language understanding. As an example, let us consider again the use of pronouns. In Section 1, we described some of the knowledge involved in choosing referents. It included syntax, semantic categories, and heuristics about the structure of discourse.

But all of these heuristics are really only a rough approximation to what is really going on. The reason that the focus of the previous sentence is more likely to be the referent of "it" is because a person generally has a continuity in his conversation, which comes from talking about a particular object or event. The focus (or subject) is more likely just because that is the thing he is talking about, and he is likely to go on talking about it. Certain combinations of conceptual category markers are more plausible than others because the speaker is probably talking about the real world, and certain types of events are more sensible in the real world. If we prefix almost any sentence with "I just had the craziest dream..." the whole system of plausible conceptual relations is turned topsy-turvy.

If someone says "I dropped a bottle of Coke on the table and it broke.", there are two obvious interpretations. The semantic categories and the syntactic heuristics make it slightly more plausible that it was the bottle which broke. But consider what would happen if we heard "Where is the tool box? I dropped a bottle of Coke on the table and it broke." or "Where is the furniture polish? I dropped a bottle of Coke on the table and it broke." The referent is now perfectly clear -- only because we have a model of what is reasonable in the world, and what a person is likely to say. We know that there is nothing in the tool box to help fix a broken Coke bottle and that nobody would be likely to try fixing one. It would be silly to polish a table that just got broken, while it would be logical to polish one that just had a strong corrosive spilled on it. Of course, this must be combined with deductions based on other common sense knowledge, such as the fact that when a bottle breaks, the liquid in it spills.

Even more important, we try to understand what the speaker is "getting at." We assume that there is a meaningful connection between his sentences, and that his description of what happened is probably intended as an explanation for why he wants the polish or toolbox. More subtle deductions are involved here as well. It is possible that he broke the table and fixed it, and now wants the polish to cover the repair marks. If this were the case, he would almost surely have mentioned the repair to allow us to follow that chain of logic.

Our system makes only the most primitive use of this sort of deduction. Since it keeps track of when things have been mentioned, it can check a possible interpretation of a question to see whether the asker could answer it himself from his previous sentences. If so, it assumes that he probably means something else. We could characterize this as containing two sorts of knowledge. First, it assumes that a person asks questions for the purpose of getting information he doesn't already have, and second, it has a very primitive model of what information he has on the basis of what he has said. A realistic view of language must have a complex model of this type, and the heuristics in our system touch only the tiniest bit of the relevant knowledge.

It is important to recognize that this sort of interaction does not occur only with pronouns and explicit discourse features, but in every part of the understanding process. In choosing between alternative syntactic structures for a sentence, or picking between multiple meanings of words, we continually use this sort of higher level deduction. We are always basing our understanding on the answer to questions like "Which interpretation would make sense given what I already know?" and "What is he trying to communicate?"

References for lecture 2

M.A.K. Halliday, Language structure and language function, in John Lyons (ed.), *New Horizons in Linguistics*, Pelican, 1971.

Carl Hewitt, Procedural embedding of knowledge in PLANNER, Proceedings of the Second International Joint Conference on Artificial Intelligence, London, 1971.

M. Ross Quillian, Semantic Memory, in Minsky (ed.) *Semantic information processing*..

Roger Schank, Identification of conceptualizations underlying natural language, in Schank and Colby (eds.), *Computer models of thought and language.*

Gerald Sussman, T. Winograd, and E. Charniak, Micro-planner reference manual, MIT AI-Memo 203, 1970.

Terry Winograd, *Understanding natural language*, Academic Press, 1972

Terry Winograd, A procedural model of language understanding, in Schank and Colby (eds.), *Computer models of thought and language*,

III Lecture 3

REPRESENTATION:

FORMALISMS FOR KNOWLEDGE

This lecture will move away from talking about natural language systems directly. It will review a number of the methods which have been explored for representing knowledge in a computer, as a background for understanding current problems in natural language. The word "knowledge" is intentionally vague, since the issues are very general and apply to many different sorts of knowledge. I will try to clarify it through showing specific examples.

Basic issues of representation

In designing a system for representing knowledge in a computer program, there are a number of issues we must face.

First, we must be concerned with how the system will make use of the representation in *operation*, and in particular we want it to be efficient. We must be concerned with the way the efficiency changes with the amount of knowledge in the system. Some representations are good for small amounts, but explode in an exponential way as r [illegible] nformation is added. Other representations are less sensitive to size, and large s [illegible] run as efficiently as small ones.

The next issue is *learning* -- the addition of new knowledge. It is important that the knowledge be *modular*. We should be able to add new facts without worrying in detail about how they connect with others. The easiest form of learning would take place in a completely independent system where each fact served as its own module. Learning would then be a simple process of accumulation. In any realistic system this is not the case, since we have to be concerned with the interactions between the new piece and the ones that were functioning before. We also want the representation to be *natural*, so that it is easy for a person to add new knowledge. If the format is difficult for people to work with, it makes it harder to put knowledge into the system.

Finally we must be worried about *building* the system. We must choose between complex structure of many parts, or a structure operating in a simple uniform way. There is a tradeoff between the complexity of structure and the generality of the system. We would like it to be able to handle as many different kinds of knowledge activities as we can.

As in much of computer science, there is no way to maximize all of these criteria, but we must look at the tradeoffs. In trying to make something more efficient, we make it less general. Making things more natural to express may demand a more complex system to use them. The most convenient way for people to express many kinds of knowledge is natural language. As we have seen in the previous lectures, information in that form can only be handled by a very complex system. Machine-language programs are probably the most efficient representation for machine use, but are unnatural, difficult to write, and usually very non-modular. Our problem is to pick a representation which combines the best features for a particular use.

I am primarily concerned with artificial intelligence problems, so the tradeoffs are somewhat different than they might be for something like production programs for business computing. In that environment, efficiency becomes much more important. In the context of research in natural language we must avoid exponential growths of running time with respect to size, but things like naturalness are much more important than details of efficiency in a system which is undergoing continual development and change.

Using a representation

A number of AI representations have been developed for use in a variety of problem tasks. This lecture will present a number of them and discuss the ways in which they give useful ways of operating on knowledge structures. Figure 3.1 lists some of the different operations a representation must support to be useful in a system.

CONTROL
    What should I do next?

RETRIEVAL
    What knowledge might I try using?

MATCHING
    Does it apply? How?

APPLICATION
    What can I conclude from it?

Figure 3.1. Operations of a knowledge-using system.

The first question is that of *control*. Given a set of facts and procedures, how does it decide what to do next at any point. There is an obvious tradeoff between tight control (which gives efficiency) and a flexible control structure which gives more generality. The more freedom there is in deciding what to do next, the more likely the system is to be able to handle situations not directly anticipated in building it. But this also may involve doing a lot of searching and looking around.

The *retrieval* process involves sorting out large quantities of knowledge to decide which ones are relevant to what is being done. In ordinary programs, this is not an issue. A subroutine is called explicitly, so there is no need to look around at others, or decide whether it is the one to use. In heuristic search, on the other hand, it becomes important to be able to choose a particular set of methods to be tried on a problem. When we try to decide which methods will possibly work, we must use some sort of retrieval mechanism.

The *matching* problem involves looking at a particular method and seeing how it actually fits with the problem. It is more specific than retrieval, which generates plausible choices, in that it is concerned with understanding just how the one chosen interacts with what is being done -- how does this program fit with the job at hand, or how does this particular fact answer the question which is being posed.

Finally, we must *use* the resulting match to draw a conclusion or have an effect. This, like the terms above, will become more clearly defined as we go through some examples.

Predicate Calculus

Let us begin with a representation from mathematics and formal logic -- the predicate calculus. In this system, a small number of possible structure types are used in a very general way to describe knowledge without regard to the particular domain. I will not describe here the details of this formalism, which are available in many places.

Simple facts, like "Fido is a dog", or "Kazuo owns Fido", are expressed in *atomic* statements like Dog(Fido) and Own(Kazuo,Fido). Quantifiers make it possible to express more complex facts like "A dog is an animal," or "Every dog has an owner." as:

$\forall x\ \mathrm{Dog}(x) \supset \mathrm{Animal}(x)$
$\forall x\ \mathrm{Dog}(x) \supset \exists y\ \mathrm{Owner}(x,y)$

Through use of a small set of logical manipulations, these facts can combined to answer questions and solve problems. If we know that Fido is a dog, then the question "Does anyone own Fido?" might be answered directly if information about his ownership were in the system, but if not it could still be answered "Yes." by deducing from the general fact above that he must have some owner. We could derive this in the form of a logical proof of the owner's existence.

There is a straightforward connection between asking questions and finding solutions to problems. It is so simple as to be a trick. Faced with a problem, such as building a certain structure out of blocks, I can phrase a question like "Is there a series of possible actions whose end result is the desired arrangement?" In order to answer this, the system will usually operate by actually figuring out what the sequence of steps must be. This is not a logical necessity, as it could know that there is a possible one on more abstract grounds, but due to the way the axioms are put into such systems, they can generally only find the proof by constructively working out the sequence of operations. will use the phrases "solve problems" and "answer questions" interchangeably to represent a kind of reasoning operation which begins with a set of facts and procedures, and ends with a desired result.

In using predicate calculus, we must have some way to generate a proof. The system must have some way to decide which fact it should apply, and to see how that fact applies to the question. Faced with "Does anyone own Fido" it must decide that the general fact $\forall x\ Dog(x) \supset \exists y\ Owner(x,y)$ is relevant. It must match the "X" in that fact to Fido, and must use rules of logic to combine this with the fact Dog(Fido) and draw the necessary conclusion.

In most systems using predicate calculus, this is all done by a *uniform proof procedure*. There is a method built into the system for taking a group of axioms and looking for a proof. The methods used can be shown to be complete. If there is enough knowledge in the system to prove something, most theorem provers will eventually get to it. But this is a significant "eventually". The demands of generality make these systems inefficient in a combinatorial way. If the number of facts is doubled, the running time is squared.

Retrieval -- the decision of what facts to look at next -- is not dealt with as a separate problem. Those things which are tried include any facts which might fit in accordance with the rules of logic. There may be heuristics which involve choosing shorter facts first, or the like, but there is nothing in the basic principles of predicate calculus or in the implementation of current systems, corresponding to the human decision of "What kinds of things seem likely to be most relevant?"

Matching is handled by a process called *unification* which matches objects to variables, in a purely syntactic way.

Finally, the application of any particular fact is to deduce a new fact or establish a truth or falsity. Each "step" involves combining some old facts according to rules of logic, to either establish new one or find a contradiction.

The advantages of predicate calculus systems are along the dimensions of modularity and generality. Each fact is valid, independent of whatever else is in the system. The notation is explicitly general, not tailored to any particular sort of knowledge. The main problem with this approach is efficiency. All of the systems which have adopted it have been limited to very tiny sets of facts, usually on the order of less than a hundred, and often less than ten. The complexity of such a system depends on how much concern there is for efficiency. In principle, theorem proving could be done with a very simple system, but the ones which have been designed are quite complex due to needs of reducing some of the gross inefficiency.

Along the dimension of naturalness, it is a matter of taste. Some people (usually trained in mathematics) find predicate calculus a very natural way of expressing things, while others find it quite difficult.

Simple programs

A very different sort of representation is the simple form of what we think of as *programs*. In programming there is a separation between *program* and *data*, as opposed to the more uniform representation of predicate calculus. The knowledge of a specific domain will be a combination of special procedures, and specific data. A program which calculates astronomical orbits will contain much of its knowledge about astronomy in the program which performs the calculation, while other will be in the form of data in various constants, arrays, etc.

This form of representation implies very different tradeoffs. The control is completely explicit. Which piece of knowledge will be called at any particular time is determined in advance by the programmer. If a procedure has some question to be answered, it contains a specific call to the subroutine which can generate that answer. This is very different from the general sort of retrieval in predicate calculus where any fact which matches the one being looked for may be used by the system, and may be added without explicit programming to call it.

The binding of arguments can be viewed as a kind of matching procedure, where the particular case is put in correspondence with a general formula. A routine says "For any number X, I know how to square it." To answer the specific question "What is the square of 3.14158?" the system *binds* the value 3.14158 to the variable X, then runs the procedure. As a result of applying a procedure, a specific sequence of further procedures might be called as subprocedures.

One of the main goals in this lectures is to show the ways in which activities like predicate calculus theorem proving and numerical calculation are really doing very much the same thing. Although knowledge that "All dogs are animals.", and "To square a number multiply it by itself." are represented very differently, they have much in common, and in AI, we are looking for the right specific tradeoffs to handle as many kinds of knowledge as possible.

The efficiency of programs is the greatest we could expect. There is no time wasted in deciding what to do next, or trying different possibilities. As programs get larger, as long as they are well structured, they do not lose efficiency, and can include great amounts of specific knowledge. On the other hand, their modularity is often bad. Structured programming is an attempt to get away from this, but in general a change to one subroutine can have far reaching effects on the others that use it. If I have a program which calls a subroutine, and change that program, then when the subroutine is called, the environment may be different from what was anticipated, and this may cause it to fail. Whenever I make changes to one thing, I must worry about how it interacts with others.

Naturalness is also not a property of programs for most domains. It is much easier to say "All dogs are animals." than to write a program which uses this knowledge effectively. The complexity of the system (the general system for manipulating knowledge) would be that of the compiler or interpreter. In both complexity and generality, there is a wide range. We can build a very complex system or a simple interpreter. We can have a general programming language or a specialized one. So choosing a *procedural* representation leaves many of these questions open while giving advantages in efficiency, and suffering in generality. It can only do those things which were planned explicitly in organizing the knowledge.

We can look at programs and formal logic as being at two opposite poles. Programs are efficient at the cost of low generality, while representations like predicate calculus are very general at the cost of low efficiency.

Planner-like languages

One of the main developments in AI has been the invention of programming languages which give us some of the benefits of a more flexible representation. They want to keep the efficiency and runnability of programs, avoiding the problems of general search, while breaking loose from some of the rigidities of program control. One such language is Planner. There is a whole set of Planner-like languages, such as: Micro-planner, an implementation of a subset of its ideas; Conniver a close descendant of Micro-planner, and QA4-QLISP, a very similar approach developed at the Stanford Research Institute.

The basic idea of these languages includes having a data base of primitive *assertions* much like the simple assertions in a predicate logic system. A simple fact like "A is on B" is represented in a data structure like: (ON A B). There is then a set of *consequent theorems* and *antecedent theorems* embedding more complex knowledge, like those in Figure 3.2.

```
(CONSEQUENT (X Y Z) (ON ?X ?Y)
        (GOAL (ON ?X ?Z))
        (GOAL (ON ?Z ?Y)))

(CONSEQUENT (X Y) (ON ?X ?Y)
        (GOAL (CLEARTOP ?Y))
        (GOAL (GRASP ?X))
        (GOAL (MOVE-TO ?Y))
        (GOAL (LET-GO-OF ?X)))

(ANTECEDENT (X Y) (ON ?X ?Y)
        (ERASE (CLEARTOP ?Y)))
```

Figure 3.2. Some theorems in a Planner-like language.

The first says that in order to establish that an object X is on an object Y, this can be done by establishing that X is on some object Z and Z is on Y. This same fact might be represented in a simple logic formalism as:

$\forall x,y,z\ (On(x,z) \wedge On(z,y)) \supset On(x,y)$

Rather than simply stating that fact, the Planner theorem states a partiuclar sequence of actions to be taken if there is a goal of establishing the fact (ON X Y). In a natural way, this can be use to describe complicated sequences of actions, as in the second theorem of Figure 3.2. It says that to put X on Y, we need to clear off Y, then grasp X, move to a location on top of Y and let go. I have included this simplified example to illustrate how Planner tries to bridge the gap between the program world and the logic world. The first theorem is much like a logical statement -- "If A is true and B is true then C is true." The second is much more like a program with calls to subroutines.

Planner-like languages use *pattern directed invocation* to make both of these work. Rather than calling a subroutine by name, a Planner theorem specifies a pattern of the result to be achieved, like (ON A B). The theorems are stored with a special index which can decide which ones match the goal pattern. When a particular goal is set up, the system automatically tries the various theorems which are indexed as being useful for this goal. The theorems of Figure 3.2 would be called for any goal of the form (ON ? ?) where the question marks indicate arbitrary elements. Planner gives the ability to be explicit in controlling what will be called by adding a *recommendation list* to the goal. This can specify a particular routine, or provide heuristics for choosing among several. If you give specific recommendations, Planner operates like any other programming language. If you don't, it provides a very general search procedure, using a *backtrack control structure* to do a depth first search of all the possibilities.

Planner also contains *antecedent theorems* which act like interrupts on the assertion of new data structures. The theorem at the bottom of Figure 3.2 says "If you ever add a fact saying that some object X is on an object Y, then also erase the fact that Y is clear on top." An antecedent theorem can specify an entire sequence of actions, and call any other sort of theorems in doing it.

The consequent/antecedent distinction is much like the notion of top-down versus bottom-up control. In the consequent case we say "Here is what I want, go try to do it." while in the antecedent case, the message is "Here's what I've just found, what can you do with it?"

The retrieval system for these languages is straightforward. In deciding what theorems are relevant to a goal or new assertion, the system calls on a syntactic pattern-matcher, comparing the form of the new item with the patterns stored in the index. That matching process also does the variable bindings (unification). The application of any fact is the running of the sequence of Planner statements in its body -- the result of calling a piece of knowledge is to explicitly direct the flow of the computation..

Production systems

Another pattern-based representation is the *production system* developed by Newell and Simon at Carnegie-Mellon University. A body of knowledge is represented by a linearly ordered set of rules called *productions* which operate on a *short term memory* of patterns. These correspond in a loose way to the theorems and assertions of Planner. A production rule is very much like a Planner antecedent theorem. The action of a production is essentially "If the patterns in the short term memory match the indexing pattern of the production, then do the actions specified in the production." Figure 3.3 shows a possible short term memory for a simple blocks world, and a set of productions to work with it. The patterns on the left of the arrow are those that trigger the production, those on the right represent its action.

Short term memory

```
(ON A B) (ON B C) (*GOAL (ON A C)) (LOCATION C (100 200 100))...
```

Productions

```
(ON X Y) (ON Y Z) → ADD-TO-STM (ON X Z)
(*GOAL(ON X Y)) (ON X Y) → ADD-TO-STM (GOAL-COMPLETE (ON X Y))
  ....
```

Figure 3.3. A hypothetical production system for the blocks world.

There some basic important differences between the operation of a production system and a planner-like language. First, the production patterns match against the entire short term memory, not just a single assertion. Therefore a production can be triggered by a combination of facts in a way which is very awkward for Planner. Another important difference is in the way of deciding which production to apply. In Planner there are mechanisms for explicitly naming the theorems, for putting in arbitrary programs as recommendations for selecting them, and a default mechanism for doing a complete search. In a production system, there is an ordering built in among the productions, and the system uses this permanent ordering to decide which production should be applied in the case that more than one is possible. Most of the work that has been done has used a simple linear ordering of all the productions. Newell and Simon have tried to show how this sort of ordering can explain many aspects of human problem solving. It remains to be seen how this will work for complex problems involving large amounts of knowledge. If it does, it provides a specific compromise between efficient but inflexible call, and general search.

There is no separate mechanism for retrieval in a production system. The decision of what to try is based on a syntactic match between the patterns in the short term memory and the patterns of the productions. Matching is done in a very simple way (intentionally avoiding the complexities of other matchers in order to remain more plausible as a psychological model for a primitive operation). The action of a production is an explicit sequence of operations on the short term memory. This is different from a planner-like language, in that a production does not directly call another productions in the same way that a theorem can call another theorem. All it can do is leave the short term memory modified in such a way that it will cause other productions to be called when the next round of pattern matching is done.

Merlin

Another system being developed by Newell is Merlin. It is in a very early stage of development, and only one very sketchy paper has been published on it. I won't go into much detail, but want to mention it because it has many ideas in common with the kind of system I will describe tomorrow, which could be worked out much more fully.

The primary data object in Merlin is a *beta-structure* like those in Figure 3.4 . The first says that a man is an animal, further specified as having a house and a nose. We view each object as an instance of some more general class with some *further specification*. A pig is also an animal, but with different further specification. The basic operation is something called *mapping*. It can be thought of as "Try to view this object as an instance of that object". If we ask "Try to view a man as an animal." Merlin will answer "It is one, with a house and a nose." If we ask "View a pig as a man", it says "I can only do that if I view a sty as a house and a snout as a nose." It then recursively asks "How can I view a sty as a house." Presumably sty will have been defined as a "house for a pig", so this mapping succeeds, and so on.

```
MAN  [ANIMAL : HOUSE NOSE]

PIG  [ANIMAL : STY SNOUT]
```

Figure 3.4. Beta-structures (Newell).

The details of this operation leave much to be worked out in terms of the selection of elements in the beta-structure for mapping, the control of which will be tried when, and what level will be taken as satisfactory, etc. But what is important is the basic idea that we should think of controlling a problem solving procedure in terms of mapping. We should look at a particular set of facts as an instance of some more general object, and the basic reasoning process involves trying to establish the correspondence in this mapping.

In a natural language understander, we might have a beta-structure to represent a particular kind of story where a person searches for a treasure and finds it. We might take the sequence of lines of some particular story we are reading and view it as a further specification of our general treasure-story. In doing this, there will be a top-down search to map the elements of the general story onto the specific events. I will talk about this kind of approach much more in the next lecture.

Actors

Another approach which is not very far developed, but represents a way of thinking about knowledge is *actors*. Merlin says that the fundamental operation is a kind of analogy. One pattern is viewed as representing another pattern, and the analogy must be established between the internal elements of them. The thought process is driven by a kind of inference based on trying to apply a stock of general descriptions to the specific data on hand, and as a result coming out with further specific facts.

Hewitt's actor formalism is a very different way of looking at things, in which everything is viewed as a procedure. So in addition to obviously procedural things, like a function definition, he also sees the number "3" as a procedure, or the list (A B) as a procedure. This seems counter-intuitive at first, but provides a kind of uniformity to all the knowledge in the system. In ordinary programming, we view data as a set of objects to be operated on. So the number 3 can be operated on to produce its sign, or magnitude, or to add it to something. If the data is a list of two elements, then we can find the first element of it, etc. In the actor way of thinking, each entity is an independent procedure which can receive messages from other procedures and send messages back. So the list (A B) is a procedure which accepts the following messages:

1. I want to know your first element. (CAR)
2. I want to know the list which is all but your first element. (CDR)
3. Are you exactly the same element as this one. (EQ)

Instead of thinking of primitive operations on data types, we think of a primitive set of messages which will be accepted by any actor which represents an object of that type. If you send a message to an actor which does not know how to handle it, it will cause an error.

This viewpoint gives an interesting perspective on ordinary programming operations like PRINT and PLUS. In the usual way of thinking about programs, there is an operator named PLUS or "+" which knows enough about different data types to be able to add them, doing the necessary conversions, calling the appropriate hardware instructions, etc. If we add a new data type, PLUS must be changed to handle it. In the actor way of thinking, "+" is a message which can be passed to any actor. The actor associated with 3 knows how to handle such a message. In the case of a binary operator like "+" this involves negotiating with the other operand about which knows how to handle the other. In a simple unary operator like PRINT we can view the message as being "Print yourself," and the knowledge of what form should be used for printing each sort of data is distributed through the system in the programs of these actors. Our list actor above would have a procedure of the form:

1. Send the message "print yourself" to the actor "("
2. Send the message "print yourself" to the actor you have as your first element.

etc.

This view is uniform, and raises many interesting questions about the way knowledge is distributed in the system. On the other hand, it is often a less natural way to look at what is happening than a more traditional view of program and data.

Looking back to the predicate calculus formalism above, it is very non-actor like. We do not think of a formula in terms of what it does, but in terms of its structure which can be operated on. On the other hand, the simple programming view is much more actor-like, and the actor formalisation is a generalization of the programming viewpoint.

Actors are not a developed system with specific choices about how such operations as retrieval and binding should be done, but much more a way of looking at knowledge, and thinking about how control should be organized. In the frame representation described in the next lecture, I will show where some actor-like ideas are being applied.

Semantic nets

Another type of representation used in natural language programs is *semantic nets*. These have been formulated and used in a variety of ways, and I will just try to point out the basic emphasis. Nets are used to express much of what we might call "common-sense knowledge". They are not designed like symbolic logic to express complex formulas and connections, but rather are a natural way of expressing simple relationships. Figure 3.5 shows a simple net. There is a node for "dog", and one for "animal" connected by a link I call "isa", indicating that a dog is a kind of animal. In the predicate calculus formalism, this fact would be stated in a quantified formula:$\forall x$ **Dog(x)** $\supset$ **Animal(x)**. For certain kinds of information, such as the class-subclass hierarchy for types, the net notation is a natural and simple way to describe things.

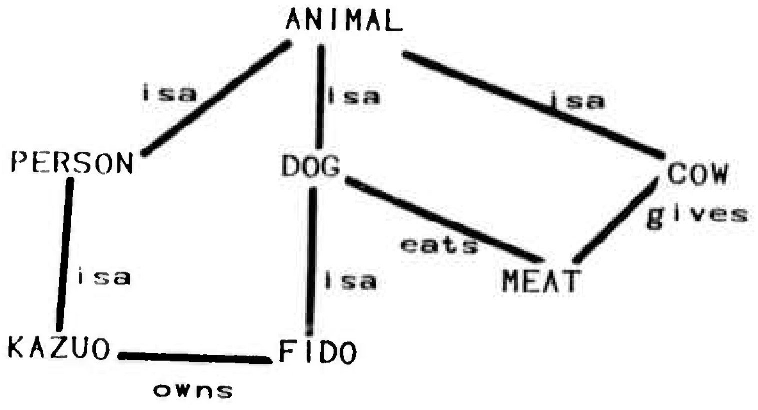

Figure 3.5. A semantic net.

Once the information is in this form, there are two basic operations that can be done on it. One is a simple kind of deduction. If we ask "Does Fido eat meat?" A system could have a set of procedures for looking at the net, and seeing the two connections "Fido isa dog" "Dog eats meat" and answering "Yes." The system would have built into it the deduction rules appropriate to the different kinds of links. This sort of mechanism has been proposed as being close to a psychological for human deductive mechanisms.

The other operation is a kind of search, using intersection in the net. It is used for deciding which links are relevant to what is being asked. If I say something about "Fido" and "meat" in the same sentence without explicitly mentioning any connection, this network could be used to find one. We can imagine sending out signals from those two nodes, spreading through the net one link at a time. When two signals intersect, the path between them will be the shortest set of links connecting the two objects -- in this case we would find the two link connection "Fido isa dog" "Dog eats meat."

One problem with this search is that it explodes very quickly as the number of links goes up. If the search must extend farther than one or two links, there will be a host of connections, some relevant and others irrelevant. For a node like "dog" or "meat", the number of connections may be very large. The ones which will be found depend strongly on the kinds of links allowed, and it is not clear how far a simple network algorithm can be extended into complicated domains.

The other problem is that of expressing more complex facts, like those involving quantifiers. The fact that "Every dog is owned by some person." cannot be simply expressed by linking the nodes for "dog" and "person", since that would not distinguish it from "Every person owns a dog." or "some people own dogs." In order to use nets well, we will need to combine their naturalness and simplicity for simple cases with the more extended power of other representations, including the equivalent of variables and quantifiers.

Semantic networks are the only representation I have described which concentrate on the problem of retrieval -- how to find the set of facts relevant to a given problem. The others have concentrated much more on how to apply the facts when they are found. The two ideas might well be combined, since the strength of network systems is more in finding connections than in making use of them.

Frames

I hope it is clear by this point that there are many issues to be dealt with in choosing a representation for knowledge. We would like to combine the benefits of all of the current approaches. The uniformity or generality of some approaches needs to be combined with the efficiency and simplicity of others. I am in the process of working out such a formalism for use in programs for understanding natural language. I would like it to be general enough to represent the facts about the world being discussed, the meaning of sentences and phrases, and also the facts about the language. It must be able to handle both those things whose general nature is best thought of as a procedure and those which are better thought of as a set of independent facts which must be worked on and put together. There are currently many people looking into formalisms which they call *frames*. It is important to point out that this does not involve an actual working system, or even a coherently worked out set of ideas. Minksy and others at MIT are working out a "frame" way of thinking for visual information processing -- for programs which look at a scene and recognize what is in it. I have been working with Dan Bobrow at Xerox PARC, developing a frame-like notation for use in natural language processing.

At this point, all we have is a collection of ideas to be developed in the coming years. In the next lecture I will go into some detail on some of the things which should go into frames, in an area somewhat abstracted from natural language. Here I will just point out some ways in which the frame viewpoint contrasts to the representations described above.

One belief is that no simple system with only a few kinds of objects and operations will be able to find the right tradeoff between all of the issues of representation. The price of a system which is really general is a fair amount of complexity in the kinds of objects and operations it includes. This allows it to operate in a natural way -- simply for simple things, but with enough mechanism to handle whatever degree of complexity is needed.

The basic object in the system is a *frame*, which can be thought of as a collection of facts and procedures associated with a concept. It is a bit like one of the nodes in the semantic nets, or like an independent actor. It does not correspond to a "single fact" like in a formal logic representation, but is a chunking of information around a single concept. Figure 3.6 gives an example parts of the frames for the concepts "give and "pay", in a simplified form. The system contains a hierarchical network classifying the concepts as "further specifications" of others. So "give" is a kind of "act". If we know that every act must have a time and place, then we know automatically know that every "give" has a time and place. Similarly, "pay" is a kind of "give".

```
GIVE isa ACT

        ACTOR: person
        BENEFICIARY: person
        OBJECT:  physcial object

PAY isa GIVE

        OBJECT:  money
        REASON:  debt
```

Figure 3.6. Some simplified frames.

Associated with each frame is a set of *important elements* or Imps. These are labelled as being of central importance to the properties of the frame. Among the many possible facts about a concept, only certain ones will be relevant for a given purpose. A central set will be most likely to be relevant. In the case of giving, the ACTOR doing the giving, the BENEFICIARY receiving it and the OBJECT being given are of primary importance. In paying, the OBJECT is further specified as being money, and the reason (which in general is an Imp for any act) is further specified as being some kind of debt. The frame for "donate" would have a different further specification for its reason.

In addition to the hierarchical structure of "isa" links and the presence of Imps, frames have an explicit indexing mechanism for finding the facts relevant to a frame in a particular context. If we are looking for a connection between "pay" and "Friday", we might want the fact that payday is Friday. In a semantic net, there would be some sort of path through this connection. In a frame system, the indexing mecnanism should allow us to ask "Do you have anything stored under the association between these two concepts?", which would retrieve the desired fact, possibly along with others.

In addition, each frame has procedures associated with it. For example, if we are trying to decide whether a particular act is a payment or not, a general procedure might cycle through the Imps seeing if they could by filled in appropriately for the given example. Or instead, the system could have a special procedure for efficiently deciding whether some act was a payment. The procedure would be attached to the "pay" frame, and would take precedence over the more general mapping procedure.

The flow of control in a frame system can be directed by specific procedures attached to frames, or handled more generally by a mapping mechanism which tries to view something as an instance of a frame, and control the process by looking through the important elements. Retrieval is based on a separate index, or association mechanism, and matching is done more generally than with a syntactic pattern matcher. To match two elements, a kind of general mapping or analogy like that of Merlin is used. The result of applying a particular frame may be to trigger specific procedures in a top down way, or may be simply adding this new description and seeing whether it is an element in some still larger concept which applies.

I want to reiterate that this system has not been worked out in detail. The attempt is to provide a system with enough facilities to do the things which other representations allow, but to do it within a coherent framework for putting things together. Simple facts are represented in a straightforward declarative way. If specific procedures are called for, there is a way to attach them in a way which allows the control structure to move back and forth between more general and more specific processes without having to pre-decide just how each piece of information will be stored or used.

An important design criterion of this system is an attempt to avoid "brittleness". We want to have a lot of redundancy in the sense that there is more than one way to get the same thing done, so if one thing doesn't work, there is another way of trying. This is not just at the level of having multiple theorems for a particular goal but in having different levels of generality at which a problem can be attacked. There can be specific methods for it, or only a loosely structured set of facts to which a more general method applies, and the system can move freely between these levels.

The frame mechanism described in the next lecture is based not on a philosophical bias about the the form which a representation should have, but on trying to look at the issues which must be faced, to combine the advantages of a variety of representations in a coherent system.

References for lecture 3

Predicate Calculus

John McCarthy and Pat Hayes, Some Philosophical Problems from the Standpoint of Artificial Intelligence. in Meltzer and Michie (eds.), *Machine Intelligence 4*, Edinburgh, 1969, pp. 463-502.

Erik Sandewall, Conversion of Predicate-calculus Axioms, Viewed as Non-deterministic Programs, to Corresponding Deterministic Programs, *3IJCAI, Third International Joint Conference on Artificial Intelligence*, Stanford, 1973, pp. 230-234.

Planner-like languages

Daniel G. Bobrow and Bertram Raphael, New programming languages for artificial intelligence research, *ACM Computing Surveys* 6:3, Sept. 1974.

Carl Hewitt, Description and theoretical analysis of PLANNER, MIT-AI-258 1972.

Drew McDermott and Gerald Sussman, *Conniver reference manual*, MIT-AI-259, 1972.

J.F. Rulifson, R.J. Waldinger and J.A. Dirksen, QA4 - A language for writing problem-solving programs, *IFIP Proceedings* 1968.

G. Sussman, T. Winograd, and E. Charniak, *MICRO PLANNER Reference Manual*, MIT-AI-203A

Production systems

Allen Newell and H.A. Simon, *Human Problem Solving*, Prentice Hall, 1972

Merlin

J. Moore and Allen Newell, How can MERLIN understand?, in Gregg (ed.) *Knowledge and Cognition*, Lawrence Erlbaum Associates, 1973.

Actors

Carl Hewitt, A Universal modular ACTOR formalism for artificial intelligence, *Third International Joint Conference on Artificial Intelligence*, Stanford, 1973, pp. 235-245.

Semantic nets

M. Ross Quillian, Semantic Memory, in Minsky (ed.) *Semantic information processing*.

David Rumelhart and Donald Norman, Active semantic networks as a model of human memory, *Third International Joint Conference on Artificial Intelligence*, Stanford, 1973, pp. 450-457.

Robert Simmons, Semantic networks: their computation and use for understanding English sentences, in Schank and Colby (eds.), *Computer models of thought and language*,

Frames

Marvin Minsky, A Framework for Representing Knowledge, MIT AI Memo 306, June 1974.

Lecture 4

FRAMES: SOME IDEAS FOR A NEW FORMALISM

As I have said several times, the basic problem in any AI program is how to take different sorts of knowledge and represent them in a way the computer can use and manipulate. In looking at a difficult problem like language understanding, there are many different kinds of knowledge, and I have been looking at a simpler area. In an area which is a bit easier for me to understand, I am seeing what knowledge is necessary, and trying to develop formalisms which could ultimately be extended to understand language.

What I will describe at first today is a simple kind of programming -- we might call it automatic programming -- which makes use of a variety of information. I will try and show in detail, for one small part, how this might be represented.

The office assistant

Let us imagine a program to help with our daily office work -- one with which we could communicate in a natural way. We might think of it as an assistant. I want to make it clear that this is not a program which I have written, but an experiment in thinking about what the problems would be.

I might have a dialog with my assistant, beginning by asking: "Print a day plan for tomorrow." The assistant does not know what that means, and asks "What is a day plan?" I respond "It is a list of all the things I have to do that day." The assistant asks "In any particular order?", and I say "In the order they will happen."

That is all the information I give in describing what I want. The assistant might then prepare some kind of schedule like that in Figure 4.1 with a set of times and events. The question I want to pose in this lecture is: What does that assistant have to know in order to do this?. What different kinds of knowledge go into converting a very simple specification into a detailed program which could print out that schedule?

If we just had a schedule-writing program, it would not be flexible in the same way that an assistant would be flexible. I might want to give additional information like "I always want headings centered, and printed in red." I might say "Don't say a.m. or p.m. if it is obvious." So in listing the events, instead of saying "8 a.m.", "9 a.m.", etc. it would say "8.a.m.", followed by simply "9" "10", "11" which would be clear. I might also say "If I have two things scheduled at the same time, tell me." If you think about a simple program written just to produce a schedule like this one, it might be very difficult to make a change like one of these, because they are global changes. They do not necessarily represent a change to one small piece of the program.

```
                 Wednesday, February 6, 1974

    7:30 a.m.  Breakfast at Tiffany's
    9:00 a.m.  Audience with Queen Elizabeth
                  Buckingham Palace
   11:30 a.m.  Lunch with Pancho Villa
                  Student Union

    1:00 p.m.  Write program to solve halting problem
    1:15 p.m.  Flight to Kyoto
                  TWA-407
    4:00 p.m.  Time to sit back and contemplate ideas about
                  automatic programming
```

Figure 4.1. An output of a "typical" schedule-writing program.

In addition to being able to make changes like these, a human assistant would also be able to correct the program if it did not work correctly. For example it might be given the wrong input format. We might have mistakenly told the assistant that dates would be input in a form like 032274 for March 22, 1974, while the actual input was in the order "year - month - day." Or we might have forgotten to tell about leap year, or we might want to ask in addition to be able to print a schedule for a regular set of events, for example "Every thursday". If we think in terms of a detailed program for printing a schedule, it is not at all easy to make it do something like that. The data types may be wrong, the information may not be in the right form, etc.

Generating the schedule program

In imagining a system to do all of this, we would like it to represent the facts about what it is doing in a much more explicit and flexible way than a simple programming language. We would like it to be more like the set of facts a person would have about the topic. The representation of this knowledge is the basic goal of this project, and I will present it here, emphasizing several aspects of that representation. These will be explained in detail, but initially we can label them as the "generalization hierarchy", the notion of "description and further specification," "procedural attachment," and the "integration of different levels of knowledge."

We will proceed by looking at the kinds of operations that our assistant might do in preparing a program to write schedules. These include: Planning, detailed programming, compiling into direct computer code, running, debugging, perhaps proving that the program works, and finally explaining how it works.

If we think about current programming systems, the information for all of these processes is in very different forms. In planning we might have flow charts or block diagrams, in programming we have a programming language, a quite different representation for the concepts. Along with the program code we might have comments written in natural language, telling facts about the programs. We include declarations, which are very different from program steps. In proving facts about the program, we might use assertions (there is much such work being done today). In explaining it we might have further natural language text. I am interested in finding ways of representing what we know about programming, and about a particular program in a much more uniform way. Tomorrow I will talk about a type of system which might use this more uniform representation in helping a programmar write complex programs.

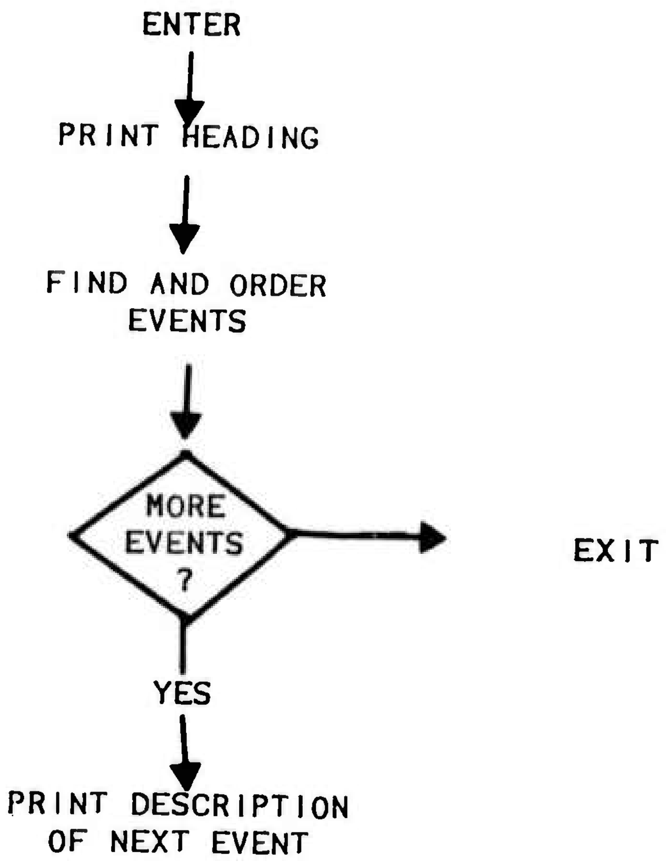

Figure 4.2. Flow chart for the schedule program.

Returning to the schedule example, the first thing the assistant might do is make a flow chart like that in Figure 4.2 . In order to do that he must know about a number of different things. For example the plan depends on knowledge about documents and the fact that they have parts like a heading and body, printed in a certain way. He has to know about lists, about finding particular objects, ordering them, and using iterative operations such as loops to do an operation to all the members of a list.

Given that plan, it would then need to be converted into a program in some programming language. If we look at what seems to be a very simple task, there are a great many different areas in which the assistant needs knowledge. In order to further simplify our example, we will look only at one small part of that -- the problem of producing the heading line which describes the date. It is given the date in some internal format and prints out a line like "Friday, March 27, 1974".

Figure 4.3 shows a program (in an imaginary ALGOL-like language) which would do the printing of the heading. There is a variety of information in the program. Declarations, for example, that the week-day and the month will be strings, while the day and year will be integers, and so on. A person writing this program might also put in comments, describing, for example, how the array of weekdays is pre-initialized, or how he has used arithmetic operations to convert a full-date string into individual components. In doing all that, he needs knowledge about the structure of dates (how weeks, days, and months are connected), about arithmetic, about printing formats, about programming conventions (how to convert things into legal statements in the programming language), about conventions for printing headings (for example the fact they should be centered) and many other such facts.

```
procedure PRINT_HEADING (integer DAY)

  string WEEKDAY, MONTH;
  integer DATE,YEAR,X,M;
  string array WEEKDAYS,MONTHS;
  comment Arrays pre initialized to names of days and months.;

  begin
    M ← DAY/10000;
    MONTH ← MONTHS(M);
    X ← DAY - M*10000;
    DATE ← X/100;
    YEAR ← X - DATE*100;
    comment Number hacking is to decode 6-digit day format.;

    WEEKDAY ← CALENDAR (MONTH, YEAR, DATE);
    comment Warning: calendar function invalid after 2001;

    printstring (WEEKDAY & ", " & MONTH & DATE & ", " & YEAR)
  end;
```

Figure 4.3. Part of the schedule-printing program.

The generalization hierarchy

One of the main things I will be emphasizing is the use of descriptions at every level of the system. The objects which are manipulated, whether they be numbers, dates, pieces of program or whatever, can be thought of best as possessing rich abstract descriptions of what they are. When we want to do some kind of operation on them, we can use the information in that description to guide what we do. In general, we might say that the basic problem of AI is the problem of finding good descriptions for objects and contexts. That is another way of wording the "representation problem".

A basic element of descriptions is the use of *hierarchies of generalization*. Given a particular object, we can have a very detailed description, or we can have a more general description which applies to both it and other related objects. For example, in this system we might have a representation for the general concept of a time period, and describe something like an appointment as a kind of time period. Other instances of time periods might include the coffee break this morning, or the entire Meiji Restoration. More specifically we might have the idea of a day. Describing today as a day gives us more information (tells us better how to deal with it) than just describing it as a time period. Similarly, we might describe a particular kind of day, like a first-day-of-the-month. We know facts about the first day of a month which apply to any day fitting that description. We might know facts about Thursdays, or about a particular day like March 22, 1974. The basic reason for putting these concepts in a hierarchy like that of Figure 4.4 is the inheritance of properties. Basically, anything which is true of all things fitting a larger description will be true of all things fitting more specific descriptions included in it. This is a first approximation. It is very useful to be able to explicitly cancel out properties. We might know that every Thursday we go to work at a certain time, but that on this particular Thursday because it is a holiday we don't go to work.

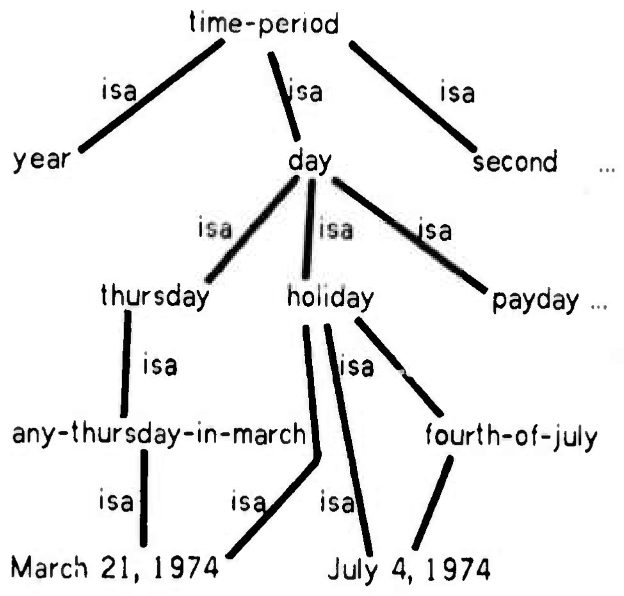

Figure 4.4. A generalization hierarchy for time periods.

In our representation, an object (description) inherits all properties of the general descriptions above it in the hierarchy, except for those explicitly marked as different for this specific instance. One other important aspect of these descriptions is that this structure is not a single tree. A particular class or instance might be generalized in more than one way. For example, March 21 is a more specific example of a Thursday, and also a more specific example of a holiday, where both Thursday and holiday are specific kinds of day. There are facts true of every holiday just as there are about every Thursday. Given any particular object, there may be a number of different descriptions, or different paths up this hierarchy which describe it. One difficult problem when we want to use an object for something is to know which description is the appropriate one. If we are interested in what we are doing on a particular day, then the fact it is a holiday is much more useful - it tells us more than the fact it is a Thursday. If we want to know where to find it on a calendar, then the fact that it is Thursday may be more important. So putting the concepts in a hierarchy leads us into a number of problems. But primarily it means that given an object, we know a lot of things about it as soon as we can classify it. As soon as we know something is a day, we have a good deal of information we can use even though it isn't explicitly mentioned.

This kind of generalization is closely related to the problem of getting very specific information (like the schedule-writing program) out of general descriptions (like "A day plan is a list of events..."). As soon as we have classified a schedule as a list, many pieces of detailed information which we know about lists can be applied. Looking at natural language, it is clear that this kind of classification is a basic part of the way we think. Given an object in the world, we almost always refer to it with a general noun -- "That is a chair." We take a general concept like "chair", and indicate the the particular object of interest is a specific example of it. As soon as we do that, our hearer knows a good deal about the object without our detailing it explicitly. The frame representation was greatly influenced by considerations of natural language and communication.

Descriptions and data types

Each of the different ways of looking at a number (like 7 in Figure 4.5) will emphasize different properties. There is a clear connection between this sort of description and the idea of data types in a programming language. There is generally a small classification scheme in a language for the different objects it can handle, and special rules for what to do to an object based on which class it belongs to. If we try to add two numbers, we know we can add them, but it we have a number and a list we can not add them. If we have two integers, we can add them with a particular kind of instruction while a floating point number and an integer take a special conversion.

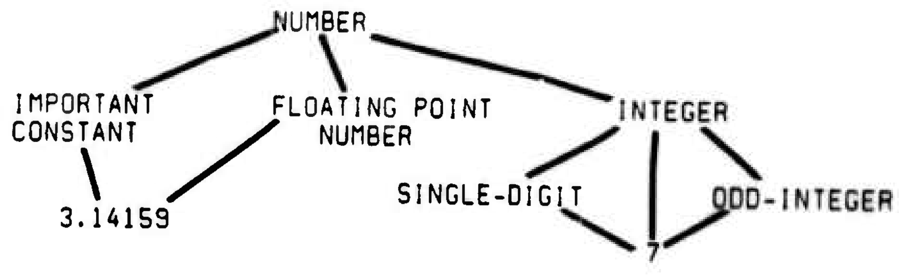

Figure 4.5. Part of a generalization hierarchy for numbers

I believe that the idea of data types in programming is too restricted. First of all, there is a simple shallow tree-like classification. We cannot classify something as both an odd number and an integer in a certain range, even though for many programs that might be useful. If we had a routine to take in the number of a month and give out its name, we would like to say that it accepts "an integer between 1 and 12." It really doesn't accept all integers, which is the typical sort of declaration available in a programming language. We have only a small vocabulary of descriptions, and no way of making better ones. We then want to extend the idea of data type to a more general notion of description.

The other problem with data types as usually done is that they are very static. You declare ahead of time just what sort of objects will be passed in each place, rather than attaching descriptions to objects dynamically and passing them around as the program runs. LISP is slightly different than this for a few examples (like the numbers in most LISP systems). In developing our example of the knowledge about a date, we will show how more elaborate descriptions can be built and manipulated.

The concept of a date

Returning to the problem of what we know about a day, we will look at the structure of the facts which a person knows about its date. This is a very particular subset of what we know -- we also know that the sun will rise, people will go to work, and so on. In this case we look at a day as a structure in a mathematical calendar system. Figure 4.6 shows some of the things we have associated with dates. The *frame* for "day" includes a set of conceptual objects associated with the date, such as the particular year and month which contain it, and the string which describes it to the computer.

DAY

year

month

day-number

day-of-week

sequence-number

ASCII-form

Figure 4.6. Important elements associated with "day".

Along with the more standard concepts, we might well have something like the ASCII form, like 740322 for use in entering data, and for some operations we would prefer simple sequence numbers which are increased by one each day. We can describe part of what we know about days in general by specifying what objects are assigned to these positions as in Figure 4.7 .

```
DAY

year                 integer

month                month-name

day-number           (integer range (interval min 1 max 31))

day-of-week          weekday-name

sequence-number      integer

ASCII-form           (integer
                       length 6
                       structure (concatenated-repetition
                                   element (integer length 2)
                                   number  3))
```

Figure 4.7. Descriptions of the IMPS for "day".

These *slots* are not filled by simple data types, but by descriptions which form part of a generalization hierarchy. So "month-name" is a particular sort of "name" which is a particular sort of "string".

We can describe any particular day as a *further specification* of this general concept of a day. Each related object would be instantiated by a more detailed description corresponding to the general one. Any object is a further specification of those things linked above it in the generalization hierarchy. This particular Thursday might be given a detailed description like that of Figure 4.8 .

Facts relating concepts

We might have a different further specification like the one shown in Figure 4.9 for "every Tuesday next April". We have further specified the year as 1975, the month as April, and the weekday as Tuesday, but we have not directly specified the other associated numbers. Conceptually, though we can further specify the day number. It isn't just an integer in the range 1 - 31. The day number for "every Tuesday next April" must be precisely an integer in the set {2, 9, 16, 23, 30}. The set is the appropriate further specification of the day number. Similarly, the ASCII form and sequence number are more specific. Our representation must be able to make these connections between those parts of the description which are a natural result of a phrase like "every Tuesday next April", and those which are true as a result of applying other knowledge, like our knowledge of the calendar.

DAY	
year	1974
month	March
day-number	21
day-of-week	Thursday
sequence-number	
ASCII-form	740321

Figure 4.8. Further specification of IMPS for "March 21, 1974"

DAY	
year	1975
month	April
day-number	
day-of-week	Tuesday
sequence-number	
ASCII-form	

Figure 4.9. Further specification for "every Tuesday next April".

In order to fit this sort of knowledge into the system, we first might put in a set of facts about dates, as shown in Figure 4.10 .

```
DAY

year              (integer
                          structure (concatenation
                                      first "19"
                                      second
                                        (! ASCII structure first)))

month             (month-name
                              (position-in-list
                                list "January February...December"
                                element (! month)
                                number (! ASCII structure second)))

day-number        (integer
                          range (interval
                                      min 1
                                      max (! month length)))

day-of-week       (weekday-name
                          (position-in-list
                            list "Sunday, Monday,...Saturday"
                            element (! day-of-week)
                            number (integer
                                      range (interval min 1 max 7)))
                          (1-1-correspondence
                            set1
                              (! day-of-week position-in-list number)
                            set2 (quotient-mod-7
                                      dividend (! sequence-number)))

sequence-number   integer

ASCII-form        (integer
                           length 6
                           structure (concatenated-repetition
                                        element (integer length 2)
                                        number  3))
```

Figure 4.10. Facts relating the Imps.

Attached to the element ASCII-form is the description that it is made up of 3 integers of length 2. This is always a possible description for an integer of length six, but it isn't always a good description. If the integer is a distance, or amount of money, the division is not useful. In this case, describing the integer as made up of three parts makes it possible to relate it to the other aspects of a date. If we think in terms of traditional programming, this kind of description never appears explicitly in the program. It may appear in the comments at the point where the program needs to take apart the integer into its components. It will appear implicitly in the existence of these operations. But the fact that the ASCII format is best thought of as 3 length 2 integers is not stated. A major goal of the frame representation is to make it easy to include this sort of description along with the data types, procedures, etc.

The year is a structure concatenating the digits "19" with the number which is the 3rd element of the ASCII form. The notation for expressing this sort of interconnection has not been worked out in detail, but as a first pass, a list is used to represent a description, with the first word giving the class (the link upward in the hierarchy), and the rest of the list giving the further specifications of the description. Lists beginning with "!" are internal path specifications which act like variables -- pointing to a particular element within the frame, to a sub-element within that one, etc. Thus the list (! ASCII STRUCTURE 3RD) represents the third element of the structure of the ASCII form of the date.

Continuing to the month, we state that there is a one to one correspondence between the position of the month in the list of the months and the 2nd component of the ASCII form. This knowledge then relates this internal format to a month name. The day number similarly is a particular component of the ASCII format, and we also know something about the range of the day number when connected to a particular month. January has numbers between 1 and 31, February between 1 and 28, and so on. This involves more complex connections. All of this is the kind of knowledge a person has about the calendar which goes into building up a program.

Finally there is a one-to-one correspondence between the position of the weekday name in the weekday list and the quotient modulo seven of the sequence number.

Procedural attachment

The facts shown in Figure 4.10 do not specify any particular way of using them. They form a static description, saying "These facts are true -- these connections hold between these abstract objects." In order to actually do something with this knowledge, like print a calendar or a schedule, we must have some way to attach procedures. To any frame or element within it, we can attach specific procedures for doing things to it. If we want to fill in the item called day-of-week, there are specific procedures to do it. One procedure is called "look up in calendar". That's what a person normally does. He knows all these facts we mentioned earlier, but if we ask "What day is June 14 this year?", he finds a calendar, turns to the page marked "June", looks for the number 14 and looks to the top of the column. This is a simple algorithm for getting the information, which does not make direct use of the declarative knowledge. The procedure associated with a set of facts may not use them directly, but may use information derived from them, or leading to the same final result.

The flexibility of the system comes from being able to make use both of the declarative facts and the procedures associated with them. If we have a procedure for looking up dates in a calendar, we can do that. It is much more efficient than calculating from the basic facts. But if that doesn't work (for example, you don't have an appropriate calendar), there are other more general procedures to try. In this case, most people have a special second procedure for finding week days which I call the use of an "anchor day". You remember the date and weekday of some special day in that month, for example, that the holiday March 21 was a Thursday. If you ask me "What day was April 17?", I calculate:

> *The 21st was a Thursday, so the 28th is a Thursday, so the 35th is..., but there are only 31 days in March, so the 4th of April is Thursday, so the 11th is a Thursday, ..Friday, ..Sat, ..Sun, ..Mon, ..Tues, ..Wed, (counting on my fingers), so the 17th is a Wednesday.*

Again we have a very specific algorithm, but one which makes more direct use of the facts, such as the fact that March has 31 days. This is an example of a more general procedure which works in the absence of a calendar -- a procedure which goes back and uses the facts.

If you said "Imagine a calendar in which February had 31 days and March had 29, what day is...", I certainly couldn't use my calendar, but I could use this same procedure by changing the appropriate facts. The loss of efficiency caused by not using a more tailored procedure is made up for by increased generality.

The attachment of a procedure to a fact need not restrict the circumstances in which it would be used. One kind of procedure might be applied when a certain bit of information is needed -- we need to know the day of the week. Another kind may be triggered as the result of learning new information. In planner-like systems, these are often distinguished as consequent and antecedent theorems, or if-added and if-needed methods, etc. It is important to cause a new fact to be deduced sometimes because we need it (top-down) and other times because we have discovered something related to it and are looking for possible connections (bottom-up).

We might have a system in which whenever we learn the number of a day, we check to make sure that it is appropriate for the month. If someone mentions the 31st of February, we would check this and be puzzled. More generally, whenever we get the ASCII form of a date, we might go ahead and fill in the year, month, and day number. We don't have to necessarily calculate other facts like the day of the week unless they are called for. The procedures provide the control -- deciding what will happen when. In operating, they refer to the factual knowledge but they add new information to it.

The important benefit of this sort of procedural attachment is the ability to integrate levels of knowledge. Faced with a question, if you have a specific procedure to get the answer, you will use it as a direct way to get the information. If there is no procedure attached, you can look up the generalization hierarchy, and see if there is a procedure for a more general concept above it. If I don't know how to find something for a day in particular, I may know something for lists in general which is applicable to one of the facts, and so on. The strategy is to look for a specific procedure, and if there is none (or it fails), you look at successively more general descriptions of what you are trying to do, and see if there is a procedure attached to one of those descriptions. At the very top, we might have a very general uniform procedure like that of a theorem prover. It has no special knowledge about how to best go about any particular task, but only weak logical knowledge about how facts can be combined. At the bottom of the hierarchy, for the most specific concepts, there will be efficient programs for special things. We must find the best level for the problem at hand, and use all of the specific facts which come to bear.

Procedural and declarative knowledge

Looking at the ideas described above at an oversimplified level, we might think of frames as combining a uniform way of expressing facts (like predicate calculus) with a detailed programming language for expressing programs, and providing a scheme to tie them together.

I would rather think in terms of making the two notations more like each other. For example in the declarative notation, rather than having only a very few connectives and quantifiers, I would say that in stating facts we need a number of different conceptual quantifiers. If we say "There exists an X with the following property..." we might have one concept for a kind of existence in which we know the precise object, and a very different kind for the non-constructive abstract existence of mathematical objects, and a further difference between knowing that there exists a single object, and that there is "one or more".

In the declarative notation used above, there are connectives like "one-to-one-correspondence". This is a common concept, but to express it in a formalism like predicate calculus can involve a quite complex statement. I would rather have a richer declarative notation, with more concepts basic to it. The cost for this is that the procedures which use this representation must have many more rules of inference. In some sense, the procedures attached at the most general level are the rules of inference, and if we have a richer set of declarative notations, we will have a fairly complicated system with special rules, for example to combine the fact that some property holds for every member of a set with some one-to-one-correspondence involving it. There is an interaction which is explicitly given rather than having it implicit in the expansion of the concept of one-to-one-correspondence into the basic connectives and quantifiers. We want a more suitable framework for representing facts -- one which is still declarative but more directly useable.

On the other end, we could think of the procedural part as being algorithms expressed in an arbitrary programming language. But if we think of AI programming languages like Planner, we see that a language can have built into its control structures many processes which are not straightforward algorithms. So at the same time we want something richer than predicate calculus as our declarative language, we would like something with more general power than, say, LISP for expressing procedures. One task in developing a representation is the creation of an appropriate procedural language.

Some connections to learning

Recently, Gerry Sussman at MIT has developed a program, called HACKER, which learns to do things in the blocks world. It faces problems like stacking one block on another, and writes its own programs to handle complex situations. It begins with very general procedures (for example a procedure for achieving two goals simultaneously) and a few very specific procedures (for example, how to put one block in a particular place). As a result of trying to do things it builds up more specific procedures based on facts it knows about the BLOCKS world.

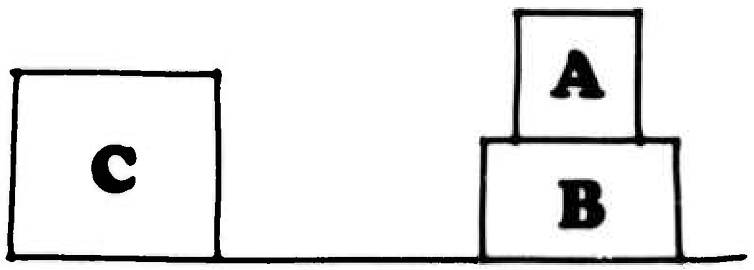

Figure 4.11. A block-stacking problem for HACKER.

Faced with the problem of putting block C on block B in Figure 4.11 , the procedure for putting blocks in places will fail, since there is something in the way. The factual knowledge about what it means for things to be in the way enables it to build a better procedure for putting things on other things, which includes code for getting rid of obstacles.

This is a kind of learning program. As a result of experience it learns to do things better in the blocks world. But it is a particular kind of learning, since all of the necessary facts were there in some form to start with, and the problem of learning is to convert them into a more useable form --- taking them in their general format and seeing how they interact with the procedures that use them. I think this is a very important view of learning which is quite different from the usual ideas of induction.

Sussman was not thinking directly in terms of the representation ideas we have been talking about, but there are some interesting parallels. If we think in terms of the generalization hierarchy, we can say that HACKER begins with the procedural knowledge attached to very general concepts, and factual knowledge at a variety of places, along with detailed procedures at a few places. As a result of learning, there are efficient procedures spread throughout the hierarchy, applying to a whole range of specific situations HACKER might encounter. In addition, we can think of induction as the process of adding new nodes intermediate in the hierarchy -- grouping a set of specific nodes by creating a more general description which applies to all of them, so information can be associated directly with that description. I do not want to imply that the problems of learning are solved by talking about them in this way, but it seems to point to some interesting avenues of exploration.

Frames for language understanding

As I said at the beginning of this lecture, my interest in developing better representations has led to looking at some specific areas which are not immediately related to language understanding like the date problem described above. For the rest of this lecture I would like to discuss how some of these ideas might be applied in a language understanding system. Much of the detail has been worked out in the context of a project I am part of at the Xerox Palo Alto Research Center. The language understanding group (directed by Daniel Bobrow) is beginning with a very simple children's story to see what problems are involved in getting a computer to understand it. The story (in its entirety) is:

> *Margie was holding tightly to the string of her balloon. A wind caught the balloon and drove it against a tree. It hit a twig and "pop". Margie cried and cried.*

It's not a very exciting story, but once we begin looking at it we see that there are many things a person must know to understand it. For example, why did Margie cry? It is not sufficient to say the balloon popped. If we said "*Margie had a firecracker. She lit it and "pop". She cried and cried.*" It wouldn't make as much sense. We might say she was frightened by the noise, but it wouldn't be the same thing. A person knows that a firecracker is supposed to pop and a balloon isn't. If a balloon pops, it is ruined. In reading a simple story a person must bring in thousands of simple facts like this to recognize the connection between the different things in the story. There is no explicit statement in the story saying that Margie lost the balloon, but we deduce it by putting together other knowledge. There is nothing saying what caused her to cry, but again we know it, and so on. Even simple things like "The wind caught the balloon and drove it against a tree." demand a knowledge of physics -- where things move and how they move, what happens when they collide, and so on. Complicated reasoning is needed in order to put together the right image which will be able to answer questions about this story.

We believe that this kind of deep understanding is necessary for any successful computer use of natural language. Even in simpler domains, and with limited tasks, like answering questions, you must deal with this problem of connecting knowledge which is not explicit.

Representing world knowledge

We are trying to apply the kind of representation ideas I have been talking about here to the facts used in understanding this story at various levels. At the deepest level, we might ask things like "What do we know about balloons?" Figure 4.12 shows some of the elements in a frame for "balloon." In one scheme of classification in our generalization hierarchy, we might have a class of "toy", and balloon is a further specification of it. From that we know a number of facts, for example that it is likely to be owned and played with by a child, that a child will like it, and so on. These follow from the general description of balloon as a toy. In addition, the frame specifies something about its shape. It is a sphere. Of course, we might have a particular balloon (say a Mickey Mouse balloon) with a specific shape which is not a sphere at all, but this does not prevent us from having

Some connections to learning

Recently, Gerry Sussman at MIT has developed a program, called HACKER, which learns to do things in the blocks world. It faces problems like stacking one block on another, and writes its own programs to handle complex situations. It begins with very general procedures (for example a procedure for achieving two goals simultaneously) and a few very specific procedures (for example, how to put one block in a particular place). As a result of trying to do things it builds up more specific procedures based on facts it knows about the BLOCKS world.

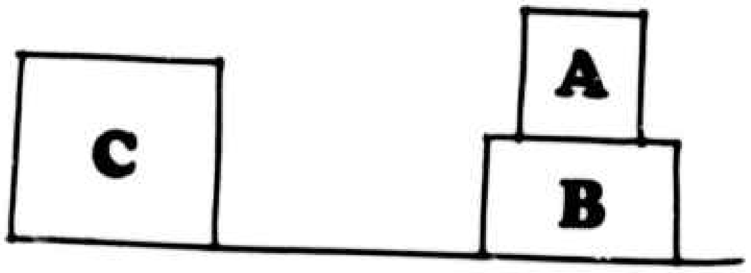

Figure 4.11. A block-stacking problem for HACKER.

Faced with the problem of putting block C on block B in Figure 4.11 , the procedure for putting blocks in places will fail, since there is something in the way. The factual knowledge about what it means for things to be in the way enables it to build a better procedure for putting things on other things, which includes code for getting rid of obstacles.

This is a kind of learning program. As a result of experience it learns to do things better in the blocks world. But it is a particular kind of learning, since all of the necessary facts were there in some form to start with, and the problem of learning is to convert them into a more useable form --- taking them in their general format and seeing how they interact with the procedures that use them. I think this is a very important view of learning which is quite different from the usual ideas of induction.

Sussman was not thinking directly in terms of the representation ideas we have been talking about, but there are some interesting parallels. If we think in terms of the generalization hierarchy, we can say that HACKER begins with the procedural knowledge attached to very general concepts, and factual knowledge at a variety of places, along with detailed procedures at a few places. As a result of learning, there are efficient procedures spread throughout the hierarchy, applying to a whole range of specific situations HACKER might encounter. In addition, we can think of induction as the process of adding new nodes intermediate in the hierarchy -- grouping a set of specific nodes by creating a more general description which applies to all of them, so information can be associated directly with that description. I do not want to imply that the problems of learning are solved by talking about them in this way, but it seems to point to some interesting avenues of exploration.

Frames for language understanding

As I said at the beginning of this lecture, my interest in developing better representations has led to looking at some specific areas which are not immediately related to language understanding, like the date problem described above. For the rest of this lecture I would like to discuss how some of these ideas might be applied in a language understanding system. Much of the detail has been worked out in the context of a project I am part of at the Xerox Palo Alto Research Center. The language understanding group (directed by Daniel Bobrow) is beginning with a very simple children's story to see what problems are involved in getting a computer to understand it. The story (in its entirety) is:

> *Margie was holding tightly to the string of her balloon. A wind caught the balloon and drove it against a tree. It hit a twig and "pop". Margie cried and cried.*

It's not a very exciting story, but once we begin looking at it we see that there are many things a person must know to understand it. For example, why did Margie cry? It is not sufficient to say the balloon popped. If we said "*Margie had a firecracker. She lit it and "pop". She cried and cried.*" It wouldn't make as much sense. We might say she was frightened by the noise, but it wouldn't be the same thing. A person knows that a firecracker is supposed to pop and a balloon isn't. If a balloon pops, it is ruined. In reading a simple story a person must bring in thousands of simple facts like this to recognize the connection between the different things in the story. There is no explicit statement in the story saying that Margie lost the balloon, but we deduce it by putting together other knowledge. There is nothing saying what caused her to cry, but again we know it, and so on. Even simple things like "The wind caught the balloon and drove it against a tree." demand a knowledge of physics -- where things move and how they move, what happens when they collide, and so on. Complicated reasoning is needed in order to put together the right image which will be able to answer questions about this story.

We believe that this kind of deep understanding is necessary for any successful computer use of natural language. Even in simpler domains, and with limited tasks, like answering questions, you must deal with this problem of connecting knowledge which is not explicit.

Representing world knowledge

We are trying to apply the kind of representation ideas I have been talking about here to the facts used in understanding this story at various levels. At the deepest level, we might ask things like "What do we know about balloons?" Figure 4.12 shows some of the elements in a frame for "balloon." In one scheme of classification in our generalization hierarchy, we might have a class of "toy", and balloon is a further specification of it. From that we know a number of facts, for example that it is likely to be owned and played with by a child, that a child will like it, and so on. These follow from the general description of balloon as a toy. In addition, the frame specifies something about its shape. It is a sphere. Of course, we might have a particular balloon (say a Mickey Mouse balloon) with a specific shape which is not a sphere at all, but this does not prevent us from having spherical shape as a further specification of our general notion of balloon. In building a system we must make it possible to allow exceptions to general rules without being confused.

```
BALLOON isa TOY

        SHAPE: sphere
        CONSTRUCTION: inflated
        MATERIAL: rubber
        ...
```

Figure 4.12. A frame for "balloon".

We also know that a balloon's construction is "inflated". Terms like "sphere" and "inflated" are in turn other concepts, each with its own important elements. For example there are important things about being inflated -- possible events like punctures or gradual deflation. They are true of all inflated objects, like air mattresses and tires, not just balloons. Therefore they are attached to the "inflated" concept rather than directly to "balloon".

We might have another frame (as in Figure 4.13) for puncture. A puncture is a kind of action involving several important elements. One is the instrument, which must be a pointed object. Another is the object, which must be inflated, and another is a result which is the destruction of the object. The notations in the figure have not been fully worked out, but attempt to deal with various problems of naming and variables. We need to say that the thing destroyed is the thing we are calling the "object". If a knife punctures a balloon, it is not the knife (or some unrelated additional object) which is destroyed. We need specific binding mechanisms so the same object can be referred to in several places. In saying that the manner of puncturing involves a collision, we must specify which objects are colliding. The names of important elements then serve two different purposes. One is just to specify that they should be considered whenever we have a puncture situation -- a kind of selection. The other is a binding for names, so we can refer uniformly to something called the "instrument" or the "object." This is like a set of variable bindings, but the exact mechanisms need to be more flexible than traditionally, and have yet to be worked out.

```
PUNCTURE isa ACTION

        INSTRUMENT: pointed object
        OBJECT: inflated object
        MANNER: (collision of (! INSTRUMENT) (! OBJECT))
        RESULT: (destruction of (! OBJECT))
        EVIDENCE: explosion
```

Figure 4.13. Frame for "puncture".

The "evidence" for a puncture would be an explosion, which we know produces a noise. In our story, there is nothing saying the balloon is punctured. It just says "It hit a twig and 'pop'." Somehow if we trigger (or activate) puncture as a potential description for the series of events we are dealing with in the story, we can then take the generalization and apply it to the event sequence. We can use the additional information associated with puncture to help understand what is happening -- in this case, using the fact that punctures cause noise to recognize what event was being described by the word "pop".

The problem of choosing the frames to try is another very open area. There is a selection problem, since we cannot take all of our possible frames for different kinds of events and match them against what is going on. We need to use clues to decide which to try. In the case of balloon, we might imagine that whenever we see one we are looking for possible things which might happen to it. One of the main facts about balloons is that they are in danger of puncture. That might tell us to be looking for event sequences fitting our puncture frame. There may be a number of mechanisms for suggesting or triggering frames.

Once a frame is triggered, it tries to fill in its important elements. In trying to find the event corresponding to the "evidence", the description "explosion" can be mapped into the noise "pop". The event of "It hit a twig." matches the description of "collision". Since the object of the collision is specified as a pointed object, twig fits perfectly. The other participant in the collision actually described is a balloon, which is an inflated object, so that part applies as well. Thus we accumulate evidence from all of these fits to decide that the notion of puncture indeed applies.

Once we have decided that, we know that the result is the destruction of the object (the balloon) even though that destruction isn't mentioned in the story. We can believe that it must have happened, since we have sufficient evidence that the frame applies, and we can use it to explain other events like Margie crying. There is a very important process here -- using partial evidence to decide that some general description fits, then using further information we know is true of things fitting that description to fill in missing knowledge. This is an approach to one of the major problems in language understanding.

Another frame we might have would describe a "loss" as an event in which there is an initial state which is some kind of possession, an object which is something of value, a person who is the owner, and a result which is the state of not having the object. A corollary of this result is the person's being unhappy. Of course this is too simplified. If you had a bad disease and lost it you would not be unhappy. In a more complete version we would want to associate the happiness or unhappiness with the value of the object to the actor along with filling in all the other connections in detail.

Given this frame, the line "Margie cried and cried." can be connected to the loss of the balloon. This demands knowing that if something is destroyed we automatically no longer have it. The loss explains the unhappiness (which we deduce from crying). In fact, the loss frame may have been triggered by noticing the unhappiness rather than deducing the puncture. Once it has been suggested, a frame tries to find matches for its important elements in the current situation.

Representing linguistic knowledge

I would like to use the same sorts of ideas for describing the linguistic structures of which language is built. As an example, we might have syntactic frames associated with words. A frame for the word "drive" might indicate how the subject and object of a sentence using it relate to the various semantic roles (cases) the objects play. It would indicate the semantic connections which are signalled by possible additional phrases like those beginning with "to" "against", and so on. If we see "I drove him to..." we know that a destination is given. If we see "I drove it against a tree.", the combination of "drive" and "against" tells us more than just the direction of motion, but also that there was a collision. The frame for the verb "drive" would be part of a hierarchy for a whole set of verbs which share properties. Since "drive" is a further specification of "move", then we want to attach the information about "to" phrases to "move" rather than "drive", since it applies to any motion verb. A "to" phrase will indicate destination for any of them. On the other hand, "against" is more limited, we don't say "He walked against the wall." to indicate a collision -- we would use "into" instead. So in the same way we have levels of generalization and description for the concept frames -- the idea of walking or moving -- we also have them for the linguistic structures -- for the way the word "drive" would be used compared to the word "walk."

We can also apply this at a kind of meta-linguistic level as in Figure 4.14 which shows a frame for the speech act of "referring".

REFER isa SPEECH-ACT

REFERENT: identified conceptual object
REFERRING PHRASE: natural language phrase
UNIQUENESS-CONDITION: reasoning chain
CONTEXT: linguistic and meta-linguistic context

Figure 4.14. Frame for the act of referring.

In language we often use a phrase to refer to some object. This can be viewed as an act in communicating to another person. The important elements are the referent (the thing you intend to refer to), the referring phrase (the words you use, which might be a pronoun like "it" or a noun phrase like "the table"), and some reason why you believe your phrase will be a unique specification to the hearer. In any current natural language system (including SHRDLU), this kind of information is implicit in the programs. There is a particular routine which handles reference, which takes in a referring phrase and tries to deduce the referent. In doing this it needs to figure out what the uniqueness conditions must be. Putting this into the terms we used earlier today, we might say that those routines have built into them the procedures which would be attached to the concepts like "referring". The specific procedures tell how to get a referent when given the referring phrase. They do not state the general facts about the process of reference. They often have totally separate sets of knowledge for finding the referent given a phrase, and producing a phrase to refer to a particular object. In the program there is no contact between these. We would prefer to integrate those places where the same facts about reference are used.

I therefore want to use frame-like representations to describe in a more explicit way the things that happen in a language interaction. Another example would be the asking and answering of a question. A program implicitly knows what to do when it gets a certain type of question. It has specific processes for doing the right thing. It would be better if we had multiple procedures attached to the notion of answering a certain kind of question, or facts which could be used by a more general procedure concerned with language interaction.

I have been describing a general framework and a few specific examples. In the course of the next few years I would like to build these ideas into a functioning system for language understanding.

References for lecture 4

John Searle, *Speech Acts*, Cambridge Univ. Press, 1970.

Gerald Sussman, A computational model of skill acquisition, MIT-AI-297, 1973.

Lecture 5

CONCEPTUAL PROGRAMMING:

APPLYING AI TO PROGRAM WRITING

This lecture presents some ideas for programming systems which are closely connected to the problems of representation we have been discussing. In the future we will be able to talk naturally about programs to the computer. We will describe them in terms which are much closer to our ways of thinking, instead of having to convert our wishes to the form most easily usable by computers. Ultimately we might just use natural language. We could speak to a computer, describing what we want done, and expect the system to understand and carry out our wishes. That is far in the future, but much sooner we will be able to program in a style I will call *conceptual programming*. By "conceptual", I mean that the act of building and using programs will be oriented much more to natural ways of thinking. We will have systems to help us with the entire range of programming activities, including: those of Figure 5.1 .

OPERATIONS

- Planning
- Writing the program
- Compiling it
- Running it
- Debugging it
- Proving things about it
- Changing it
- Describing and explaining it

INFORMATION STRUCTURES

- Graphic Aids
  - Flow Charts
  - Block diagrams
- Program
  - Declarations
  - Program Statements
- Formal Assertions
- Natural language comments
- Cross-reference listings
  - and other machine aids

Figure 5.1. Programming knowledge.

We should think of a program as a growing thing -- not some static object defined once and for all. We do not begin with a totally defined problem and write a program which might be proved "correct". Instead we often face a loosely defined problem, like the problem of providing good service to a set of time-sharing users, or the problem of understanding language, or handling all the finances for a company. These are not closed problems like that of calculating a square root. Even if there are no mistakes made in writing a program, we still need to go through a process of evolution, adapting it to new uses and unanticipated demands. My emphasis here is very different from many advocates of *structured programming*. There are many ideas in structured programming which are good, but the emphasis on structuring and proving a program assumes that the desired behavior can be described in simple terms. I am much more concerned with the wider and more interesting class of programs which need to adapt and change. The devices I am proposing are aimed at making that process more effective.

The role of a computer system

If we look at current systems, all of the different steps mentioned above are done in very different modes with different representations. The planning will be done on paper, perhaps using some flow charts and block diagrams. A programmer will then set those aids off to one side and begin writing the code. In some systems (such as advanced LISP systems), the processes of code writing, running, and debugging are integrated under the coordination of a single interpreter, making it easy to shift back and forth between these phases. However the majority of systems in use do not have this uniformity. Each stage in the process demands communicating with a separate part of the computer system, perhaps using a separate notation, and involving a high overhead in going from one operation to the other (for example having to recompile an entire program when a bug is found). Some parts of the operation are not really integrated into the system at all. In order to explain programs, a person uses natural language comments, or specialized facilities such as cross-reference listings. These provide information in a form totally separated from the rest of the program.

In the future, programming systems will center around a single interactive monitor which maintains the information for all of these phases, and lets the user move freely among them. In addition to the program it will build up a large, rich structure of *descriptions*. This concept is of prime importance. The presence of a particular sequence of code instructions is only one way of specifying a program. Other ways are necessary for other uses. We want a higher level description of the global structure -- which major pieces exist, and which others they communicate with. Another kind of useful description might be an input-output description, giving the *intention* of a piece of code. For operations like planning, explaining, and debugging, this kind of description is often much more important than the code itself. If we try to debug something, we don't initially worry about the internal structure of an errant bit of code, but what it was intended to do, and how it was wrong. This then leads into examining the code selectively. The object we now call a program is only one way of looking at what a program is. it happens to be the one the interpreter or compiler uses, but the human uses are at least as important. We don't want to think of a there being a "real" program with tangential information attached, but of a large information structure, part of which is used by the machine in execution.

In order to understand the advantage of this view, imagine that a programmer has a human assistant who works with him or her in writing programs. This assistant is not very creative, but knows basic things about programming, and is very patient and careful. Such an assistant would make use of all of the different information structures of Figure 5.1 , looking back and forth between flow-charts, comments, and code in order to help. I imagine a computer system which would operate much like such an assistant. It need not be able to analyze natural language, if we devise a special description language to convey information like that in comments. At a later date it would be good to add real natural language capabilities, but I believe that the advantages of such a system need not wait for that. In fact, having such a system could be a great help in building natural language programs.

Semantic descriptions

There is a similarity between the ideas here, and the discussion in previous lectures about developing a representation which can handle many different sorts of knowledge in a natural language system. Here I am proposing to use such a representation for the many kinds of knowledge in a programming system.

Taking this approach, we can compare specific aspects of the two areas. For example we might look at data types as semantic descriptions, rather than as purely syntactic devices. We might want to say "In this program there will be a kind of object called a SYNTACTIC NODE." (This example is from my natural language programs). Figure 5.2 shows some of the important elements associated with such a node. Each node will include a list called FEATURES, each element of which is a syntactic feature. We will have another element which is a PHRASE, which is a segment of a sentence. There will be another element called a SEMANTIC STRUCTURE, etc. In planning a program, a human programmer plans the contents to be associated with an object like this before worrying about whether it should be implemented as an array, a vector, a string, or whatever. These terms are *semantically oriented*, describing what the conceptual role of the structure is, rather than specifying its exact form.

```
SYNTACTIC NODE isa NODE

        FEATURES: list of syntactic features
        PHRASE: segment of input sentence
        SEMANTIC STRUCTURE: semantic node
        ...
```

Figure 5.2. Frame for a data object in a natural language system.

I believe that a programming system should be able to accept and make use of this sort of information. These kinds of descriptions should be given to the system during the planning phase. From the very beginning, the programmer should be dealing with the system, not working separately on paper. If we look at this description of what makes up a node, it looks very much like the frames we had yesterday for things like "balloon" or "puncture". The important elements associated with a frame are in turn each given a semantic description. So to fill out the frame in Figure 5.2 as we did for "day" in the previous lecture, we would need a similar description for SEMANTIC STRUCTURE, and each of the other concepts which go into it. The frame for SEMANTIC STRUCTURE would name its important elements, such as DETERMINER, RELATIONS, and so on. The system would already have a great deal of information about the basic data structures such as arrays, lists, and strings. In planning a program, the programmer might specify how his conceptual structures are to be implemented in terms of them, but these detailed decisions can be delayed for a much later point in the programming, and can even be based on statistics gathered by the program, or its internal knowledge of the advantages of different data types for different operations.

To carry the analogy a bit further, we might think of generalization hierarchies of these descriptions. A SEMANTIC STRUCTURE in general might have certain properties, while a further specified type like OBJECT SEMANTIC STRUCTURE will inherent those properties true of all semantic structures, while adding those peculiar to it. Similarly, a particular instance of an OSS will be a further specification of the general one, and can inherit any properties and default values not otherwise specified. Such a hierarchy will be built into the system for those concepts used in programming in general. For example, property list and alphabetized list will both be types of list, but with additional special facts of their own. Thus we can say that a particular conceptual structure in our program is to be implemented by an alphabetized list, and the system will know many of its properties (for example how to insert elements into it, or how to efficiently search it) from its general knowledge.

Declarations and high-level operations

Along with the use of descriptions as an organizing structure, we can also make use of them for declaring what is happening in the program. For example, instead of a simple declaration like "X is a list", we might say "The variable X contains a list of function names." We might have an even more detailed description like: "X is list of names of already-defined functions." In a conceptual system, this declaration is just as acceptable as "X is a list." In fact it is much better, since it provides more information for every aspect of working on a program. A declaration viewed in a broad sense is the fullest possible description of what a particular object is. We might use a similar sort of description for the input a program expects, or the output it produces. We might say of a particular function "The input must be an integer in the range -20 to +233, but not 0, and its output is a positive even integer less than 100."

The system can use these declarations in a number of ways. An ordinary compiler uses declarations to decide which operations are appropriate for which objects. If we have a "+", the actual machine operation will depend on the types of the quantities being added. Compilers also do a simple kind of checking, making sure that a procedure is not being passed the wrong type of argument or passing back the wrong type of result. These same operations can be greatly expanded in a conceptual system. We might have conceptually oriented operations, as in a statement like (ADD X to L). We are stating that an element X is to be "added" to an object L. If L is a list, the actual code for this might depend on just what sort of list it is. If L is alphabetical, then we need an insertion operation. If L is being used to represent a set, then we should check first to avoid duplicates. If L is a list of names, and X is a name of the appropriate kind, then it should be inserted. However, if X is a list of such names, they should be merged, rather than adding X as a single element.

This sort of subtlety would be simple and obvious to a human assistant. We would normally say things like "Add the new functions to the function list." or "Add COSINE to the function list." without worrying about the details of how the list is structured (in this case it would likely by a non-redundant set, probably ordered). These details cannot be filled in by a program unless it has the kind of additional description information we are proposing. If it knew that L was always supposed to contain a list of function names, then it wouldn't *make sense* to add another list as a single element, but would know (from general knowledge it has about lists) that it would be appropriate to merge them. By giving it this sort of information about intentions, plans, and concepts, it will be able to let us write in this higher-level form, one step further removed from the details of machine operation. The programmer is still in charge of describing what is to be used (we are not here worrying about things like optimization of data structures), but once we have provided basic information about what we want, we no longer have the burden of matching all of the details to it. The system can take over that task, using the information over and over.

Another high-level operation might be building structures, as in the command: (BUILD an OSS using NODE23). Again, for an assistant this would be a clear instruction. In terms of programming, this is a very loose specification. The meaning of terms like "build" and "use" are highly dependent on just what the objects are. The connections between them depend on knowing just what information is conceptually associated with an OSS, and how it corresponds to those things associated with a node. In the same way that a compiler uses data types to choose detailed operations, an intelligent system could use its semantic information to compile high-level statements into working code. This verges on what might be called *automatic programming*. If this capability were extended to more and more abstract and general commands, it would approach a situation in which the programmer describes in natural language what is to be done, and the system writes the program. The system I envision for the near future is a kind of middle ground. The programmer still states a sequence of steps, but these are much more conceptually oriented than the statements of current programming languages. The additional structure of descriptions makes it possible for the system to fill in the details.

Assertions and checking

In addition to describing what kind of things are associated with various data objects, we want to put additional information into *assertions*. Associated with a procedure there might be statements such as "At this point, the result list should be the same length as the input list.", or "After executing this statement, X should be non-zero." It would be inefficient to actually code a test for these conditions whenever they occur, since if the program is running correctly, they should always be true. An assistant would put them into some sort of information store, and refer to them if problems come up in running the program or changing it.

One of the most important uses of much of the information in the system is this sort of dynamic checking. We can imagine building our system to go into a mode where it actually makes these checks, to track down what is happening. The system would have different modes of running a program. In the most efficient mode, these assertions would not be used at all, but would be assumed true. In a more careful mode, which I call *cogitation*, these might be checked each time they were encountered. This is something a real assistant couldn't do if there were many of them, but which can be extremely valuable in debugging and changing a program. Assertions can also be used in generating code. If there is an assertion that a certain variable will have special properties at a certain point (for example it is non-nil, or negative), then the operation chosen to add something to it may be different from the one which would be chosen based on the general data type of that variable.

Assertions can be of a variety types, and I have not worked out in detail what they are, but some simple categories would include *tests*, *pronouncements*, and *intentions*. A test would say "I expect the following to be true here." We can classify test as *preconditions* (things which must be true for a particular operation to be applied) and *postconditions* (things which should be true after the operation). A pronouncement might look like: "After this step, this semantic structure is complete and will not be changed again." This is not the sort of thing which could be tested at that point -- there is no way to see whether it will be changed again. But the assertion could be added to a data base of information so if some operation tried to change that structure at a later point, it could tell that something had gone wrong. One precondition on operations which change data structures is that these structures be open to modification. Of course, as with other preconditions, this would only be checked when the system was in a careful mode looking for the source of actual or possible difficulties. We want to be able to take these extra steps when necessary, but use some other part of the description (for example the bare code steps) when we want to run efficiently.

Another sort of information would be *intentions* or *purposes*. In many current systems, there are intentions in a form like predicate calculus. For a list-sorting program, the intention would state: "As a result of running this program, it will be the case that for all X and Y in L, X is less than Y implies that the position of X in L is before the position of Y in L." Such assertions can be manipulated by standard logical operations. In the system I am proposing, intentions would be stated conceptually, and might be like: "The purpose of this piece of code is to find the referent for this natural language phrase." People familiar with current work in automatic programming will find this a strange sort of intention. There is no formal way of describing what it means to find the "correct" referent of a phrase, except in terms of the program which does it. But this does not mean that such an intention is not useful to have for a variety of operations, in describing the program, changing and debugging it. These assertions form a vital part of the set of conceptual structures built up in thinking about the program.

Some other uses of a smart system

All of the devices described above imply the presence of a very complex mechanism available for operation all the time. An AI-like program forms the basis of the system. It does not need to be as complex as, say, a natural language system, but must be fairly sophisticated in combining pieces of knowledge and drawing conclusions from them.

These same powers could be used in a variety of other operations. For example, any system will have an internal representation for the objects within the system, such as the programs and data structures. Often, if we want to look at one of these structures, we do not want to see it in the form actually stored or used internally. The form which is clearest to read may not be the most efficient. The system should be given information about how structures should be viewed, and how they will be presented. A simple example is the GRINDEF or PRETTYPRINT program in LISP, which uses indentation to show the parenthesis nesting structure. In general we would like to indicate special formats like "All of the conditionals should be in an IF...THEN form." Whenever the system prints things out for the user, the more perspicuous format is used, and on typein it can parse special structures and interpret them. This does not represent a severe efficiency loss since we are thinking in incremental terms. The system does not use a user-readable format to dump things out and read them back in. These operations would have their own formats, and the kinds of conversions needed for conceptual input-output would only take place at the rate a person would type or read, or in preparing documents. You can think of the system as simply containing all the information, and each time you sit down to use it, it is in whatever state it was left. There is no need for a concept of "filing things away" or "reading them in".

Of course internally there should be a file structure. We need to be able to group a set of programs into a conceptual file, such as a "utility file" or a "natural language file". This could involve recursive block-like structures, as well as the ability to cross-classify the same functions along different dimensions. One of the problems of structured programming is dividing up the functions in the system. For example we might separate them into data base functions, user-interaction functions, and arithmetic calculation functions. A totally different division might be into the payroll, inventory, and tax calculation parts. The same data-base function may be used in a variety of operations, and a complex operation like tax calculation may call on all of the different blocks of underlying functions. In dividing things into a simple block structure, we must choose one organization or the other. If we think of files as ways of organizing things, rather than a choice of where to store them, then there is nothing wrong with using both classifications, and allowing a single function to fit in more than one. We might say "Print out all the data base functions." or "Print out all the functions used in inventory." and no problems would result.

In debugging, one thing which should be available is the ability to pass a description along as a datum. Very often we look at a particular data object and wonder "Where did that come from?" All of a sudden a certain variable contains a strange value, and it is not clear how it got there. We might try and trap every time the variable was set to see when it got this value, and that is one kind of thing the system should be able to do in its cogitating mode. But in some cases this would not be effective. If the current value could carry along with it a description of where it came from and why it was put there, then at any point in the computation this description could be checked and the information used. Clearly this information should not be passed around all the time. But conceptually we can think of the actual value as only one particular description of the object being passed, and additional information can be added whenever it is needed. I would like to think of all data objects as complex descriptions, where for most cases, only one small part of that description is being maintained and used, but for purposes such as debugging and reasoning, other things could be added, and the actual "data" might even be left out. We should be able to do a kind of *meta evaluation* of a piece of program, passing it an abstract description of its argument, and letting it use the information associated with that description to build up a better description of what the program does without running it for a specific input. Through clever use of things like hash coding, this view need not cost a tremendous price in overhead -- we don't actually pass around an elaborate data structure which is a description, when most operations don't need it. But the information equivalent to such a description can be available at any point.

Finally, in addition to debugging, compiling, and running programs, we might use some of this information for answering questions posed by a person trying to understand the program. One strong motivation for paying the price of a complex system like this would be the help it gave to a group of people working on the same program and needing to share information. If one person declared that a particular function produced a list of names of functions already defined, another person could ask for that information in deciding how to use it or modify it. Much of the information that is now in comments for other people would be built into the conceptual structure for use both by those people and the system (and the original programmer, for times when details have faded away from mind, as they must in any large system). In addition to the specific facts entered, the system could deduce information of its own. A person might ask "What functions call data-base functions?" Thorough the use of its internal descriptions, the system could provide the answer. More complex things like "Does any function access the variable X while it is bound to this result?" are undecidable in theory, but very often can be easily answered from the stored information and information about its structure. Such facilities are gradually being built into systems such as INTERLISP, but need to be integrated and expanded in the context of the uniform information structure being proposed here.

Building the programming system

There are some basic problems which must be faced in trying to program a system like the one described here. One is clearly the need for a well-developed reasoning program. This sort of fully integrated system depends on having the ability to do the same sorts of simple reasoning about programs that a person might do. This power is necessary to make the system realistic. People will not spend time putting things like comments in programs unless they can see a direct benefit to be gained by doing so. If the system can take away the burden of much of the work by allowing such things as very high-level conceptual commands and providing answers to questions, then the programmer will be willing at every stage to give it the information which will make that possible.

We cannot then build this system until we have a high degree of artificial intelligence, but we need the programming system as tool for building AI programs. There will have to be a kind of gradual bootstrapping. The AI representations available now, like Micro-planner, Conniver, QLISP, and the various theorem provers, are not really adequate. We must build a new basis for a more flexible reasoning program. At the same time, we can be building many of the individual features I have been describing. Many already exist in systems available. There will be a cycle of putting in more and more useful parts, using those to write better reasoning programs, and gradually evolving upward to integrating the reasoning with the programming, and eventually incorporating natural language programs as a way of communicating with the system.

Another obvious problem is size. Even with all the necessary ideas worked out, we would have an implementation problem since the kind of power demanded would mean a very large computer system compared to those commonly available now. I believe this is not really a long-run problem. The costs of memory and processing are going down very rapidly. Systems supporting major research projects 15 or 20 years ago were far less powerful than mini-processors now available for a few thousand dollars. Compilers were not practical with the earliest machines, but now everyone uses them. In a similar way, complex systems like we are discussing will soon be within the reach of everyone's computational power. While other costs are going down, the cost of human programmers is going up. So people will be willing to do things which demand large amounts of storage and computation in order to make it easier for the people doing programming. The economics are in favor of much more advanced systems being practical. Also, a number of ideas are being developed to avoid really needing the size implied by the amount of knowledge in the program. Ideas such as segmentation and cache storage can be applied in both hardware and software, and we may get by with amazingly small actual machines to handle extremely large virtual size and capability. In the five to ten year future, systems should be big enough, and reasoning programs advanced enough to make many of these features possible. I should reiterate that good programming systems do not demand a super-intelligent program. We can get by with a moderately stupid assistant as long as he doesn't make mistakes. The degree of AI needed is much less than that needed for a full-fledged natural language or vision system.

I have not even begun to design a full system of the type I have discussed, and am not actively engaged in working on one. I find that whenever I begin to write a program, I say "Wait -- it seems that the system could be doing some of this work for me." and I go and work on that for a while, then return to the task at hand. Gradually, pieces of this sort of system are being accumulated, but I am not committed to building them into a form which is complete or generally useful. Hopefully, by talking about these ideas I can persuade others to implement them, so I can use the system they develop.

Where is AI research headed?

Looking at the past few years of research in AI, one sees a kind of branching path, which I believe is now coming back together. In the early stages, people were skeptical that computers could ever do anything which might be labelled "intelligent". Computers were seen as elaborate adding machines, limited to tedious calculations. The first efforts in AI were an attempt to say "Look, we can make a machine play chess! We can make a machine answer some algebra problems! We can make a machine move around in a simple environment!" There was an initial surge of effort to just get a computer to do these things. Following that, there were two different paths taken. The "Western" approach (exemplified by much of the work at Carnegie-Mellon, Stanford University and the Stanford Research Institute) was to go for more generality. Rather than working on more complicated specific problems, such as the structure of English or robot vision, people worked on general issues of representation and problem solving. Much of the work of McCarthy, Newell and Simon, etc. was in this direction. It led to much interesting work in psychological modelling, and in computational logic, which is really a branch of mathematics dealing with logic and proof procedures. However, its spirit often moved quite far away from the original idea of making programs do more and more intelligent things. The other approach (exemplified by MIT) has been to take successively more complex tasks (such as language understanding and algebraic manipulation) and look for the kinds of representations and organizational systems needed to accomplish them.

Today I see a movement towards a middle ground. The theorem-proving craze is slowing down. People are aware that very general systems are not going to be the basis of practical programs, and people who have been doing specialized programs are asking what these programs have to offer which can be brought to bear on more general problems.

Problems of "representation" are seen as central to further progress. In the next few years we will see an attempt to synthesize a more sophisticated view of generality. People are groping for an approach in which intelligence need neither be boiled down to a few simple crystals or left scattered and idiosyncratic for each problem. They are concentrating on the specific problems of representation: dealing with partial information, the flow of information and control within complex systems, ways of combining detailed knowledge of a domain with very general principles for reasoning, and ways of working in environments of uncertainty, like speech processing, in which the inputs are not reliable. They will be experimenting with system organizations which make it possible to include many sorts of knowledge, and take advantage of whichever is best suited for what is being done at the moment. This problem applies to any AI task, whether natural language, vision, game playing, automatic programming, or whatever. Each has its own specific problems, but shares many aspects with the others. In working in any specific area we look for the different sorts of representation and organization which are appropriate, and the ways in which they can combine with those from other areas.

One other trend, at least in the United States, is an increasing emphasis on aplying AI to real problems. This will involve moving away from fundamental psychologically-oriented problems like seeing and understanding, towards things with more immediate practical uses. One such area is automatic programming. This would involve methods from AI, but would be of use to anyone writing practical programs. Much work is being done in this area at the moment, partly because of a large increase in funding for it. Another area in which many applications will be developed in the next few years is medicine. A number of aspects of medical information processing are at the level where current AI techniques can just about be applied. Computers are already doing things like the analysis of electro-cardiograms. This is particularly suited to computers, since people do not begin with the advantage of a common-sense intuition about them. Doctors have to learn to read them, and the knowledge involved is very explicit and can be converted to a program directly. Other activities, like diagnosis, are more tied up with natural common-sense reasoning, but nevertheless are specific enough that in the near future a computer will be able to compete with an expert in particular disease areas, and do as well as most doctors in a variety of tasks. It is a question of economics at this point, since it is just beginning to be in the range of feasibility to use computers in this way. Also there has been a good deal of new funding available in this area.

Another area is industrial automation. You are aware that this is being pursued very actively here in Japan, and will also be studied more in the U.S. Labor costs continue to rise, and it seems that only a slight improvement in the state of the art will produce usable automation systems. There isn't a general robot worker on the horizon, but there will soon be systems which are economically competitive to take over many specific jobs.

I don't think that the general AI projects like those at MIT, Stanford, and Carnegie will end, but there will be an increasing emphasis on applications areas instead of more abstract toy problems. Work in these areas may put pressure on scientists to move away from basic issues to get results. If someone is trying to build the best robot which can be completed by next year, he will try to avoid any really hard problems that come up, rather than accepting them as a challenge to look at a new area. There will be pressure from the organization of the projects and funding agencies to get results at the expense of avoiding hard problems. Counteracting this, there is increasing attention from other scientists, such as linguists and psychologists, who are beginning to see AI as a tool for attacking the hard problems they have been facing for many years. One interesting thing to watch for will be the interplay between these goals.

In any event, it seems almost certain that the amount of research related to artificial intelligence will continue to grow at the rapid rate it has in the past few years, and that we are still just at the beginning of a major new field of human knowledge.

References for lecture 5

O.J. Dahl, E. Dijkstra, and C.A.R. Hoare, *Structured Programming*, Academic Press, 1972.

Peter Deutsch, An Interactive Program Verifier, Xerox CSL-73-1, 1973.

Ira Goldstein, Pretty printing -- converting list structure to linear structure, MIT-AI-279, 1973.

David McDonald, The LISP indexer, MIT-AI-MEMO, 1974.

David Moon, *MACLISP Reference Manual*, Project MAC, MIT 1974.

Warren Teitelman, *INTERLISP Reference Manual*, Xerox PARC, 1974.

Terry Winograd, Breaking the complexity barrier (again), Proceedings of the ACM SIGIR-SIGPLAN interface meeting, Nov. 1973.